William Stallings

Sistemi Operativi

STRUTTURA INTERNA

PRINCIPI DI PROGETTAZIONE

JACKSON LIBRI®
UNIVERSITÀ

SOMMARIO

PARTE PRIMA
ELEMENTI DI BASE

CAPITOLO 1
INTRODUZIONE AI SISTEMI DI ELABORAZIONE 3

1.1 Elementi di base 3
1.2 Registri del processore 5
1.3 Esecuzione delle istruzioni 7
1.4 Interruzioni 11
1.5 La gerarchia della memoria 23
1.6 Memoria cache 27
1.7 Tecniche di comunicazione di I/O 31
1.8 Letture raccomandate 34
1.9 Problemi 35
Appendice 1A Caratteristiche delle prestazioni delle memorie a due livelli 38
Appendice 1B Gestione delle procedure 46

CAPITOLO 2
INTRODUZIONE AI SISTEMI OPERATIVI 51

2.1 Obiettivi e funzioni dei sistemi operativi 51
2.2 Evoluzione dei sistemi operativi 56
2.3 Aspetti principali 67
2.4 Caratteristiche dei sistemi operativi moderni 79
2.5 Panoramica su Windows NT 82
2.6 Sistemi UNIX tradizionali 93
2.7 Sistemi UNIX moderni 97
2.8 I prossimi capitoli: Sommario 99
2.9 Letture raccomandate 104
2.10 Problemi 105
Appendice 2A Internet e risorse Web 106

PARTE SECONDA
PROCESSI

CAPITOLO 3
DESCRIZIONE E CONTROLLO DEI PROCESSI 113

3.1 Stati dei processi 114
3.2 Descrizioni dei processi 130
3.3 Controllo dei processi 140
3.4 Gestione dei processi in UNIX SVR4 149
3.5 Sommario 155
3.6 Letture raccomandate 156
3.7 Problemi 156

CAPITOLO 4
THREAD, SMP E MICROKERNEL 161

4.1 Processi e thread 161
4.2 Multiprocessing simmetrico 176
4.3 Microkernel 180
4.4 Thread e SMP in Windows NT 186
4.5 Thread e gestione di SMP in Solaris 193
4.6 Sommario 200
4.7 Letture raccomandate 201
4.8 Problemi 201

CAPITOLO 5
CONCORRENZA: MUTUA ESCLUSIONE E SINCRONIZZAZIONE 205

5.1 Principi della concorrenza 206
5.2 Mutua esclusione: approcci software 215
5.3 Mutua esclusione: supporto hardware 223
5.4 Semafori 227
5.5 Monitor 242
5.6 Scambio di messaggi 250
5.7 Il problema dei lettori/scrittori 257
5.8 Sommario 260
5.9 Letture raccomandate 263
5.10 Problemi 264

CAPITOLO 6
CONCORRENZA: STALLO E STARVATION 275

6.1 Principi dello stallo 275
6.2 Prevenzione dello stallo 283
6.3 Esclusione dello stallo 284

6.4 Rilevamento dello stallo ... 290
6.5 Una strategia integrata per lo stallo ... 292
6.6 Il problema dei filosofi a tavola ... 293
6.7 I meccanismi di UNIX per la concorrenza ... 295
6.8 Primitive per la sincronizzazione dei thread in Solaris ... 298
6.9 I meccanismi di Windows NT per la concorrenza ... 301
6.10 Sommario ... 303
6.11 Letture raccomandate ... 304
6.12 Problemi ... 305

PARTE TERZA
LA MEMORIA

CAPITOLO 7
LA GESTIONE DELLA MEMORIA ... 313

7.1 Requisiti della gestione della memoria ... 313
7.2 Il partizionamento della memoria ... 317
7.3 Paginazione ... 329
7.4 Segmentazione ... 334
7.5 Sommario ... 335
7.6 Letture raccomandate ... 335
7.7 Problemi ... 336
Appendice 7A Caricamento e Link ... 338

CAPITOLO 8
LA MEMORIA VIRTUALE ... 347

8.1 Strutture hardware e di controllo ... 347
8.2 Il software del sistema operativo ... 369
8.3 Gestione della memoria di UNIX e Solaris ... 390
8.4 Gestione della memoria in WindowsNT ... 397
8.5 Sommario ... 399
8.6 Letture raccomandate ... 400
8.7 Problemi ... 401
Appendice 8A Le tavole hash ... 404

PARTE QUARTA
LO SCHEDULING

CAPITOLO 9
SCHEDULING MONOPROCESSORE ... 411

9.1 Tipi di scheduling ... 412
9.2 Algoritmi di scheduling ... 416

9.3 Lo scheduling tradizionale di UNIX 440
9.4 Sommario 442
9.5 Letture raccomandate 443
9.6 Problemi 443
Appendice 9A Tempo di risposta 447

CAPITOLO 10
SCHEDULING MULTIPROCESSORE E IN TEMPO REALE 451

10.1 Scheduling multiprocessore 451
10.2 Scheduling in tempo reale 464
10.3 Scheduling in UNIX SVR4 478
10.4 Scheduling in Windows NT 480
10.5 Sommario 483
10.6 Letture raccomandate 483
10.7 Problemi 484

PARTE QUINTA
INPUT/OUTPUT E FILE

CAPITOLO 11
GESTIONE DELL'I/O E SCHEDULAZIONE DEL DISCO 489

11.1 Dispositivi di I/O 489
11.2 Organizzazione delle funzioni di I/O 491
11.3 Progettazione di sistemi operativi 496
11.4 Gestione di buffer di I/O 499
11.5 Schedulazione del disco 503
11.6 RAID 511
11.7 La cache del disco 520
11.8 I/O di UNIX SVR4 523
11.9 I/O di Windows NT 528
11.10 Sommario 530
11.11 Letture raccomandate 531
11.12 Problemi 532
Appendice 11A Dispositivi di memorizzazione a disco 535

CAPITOLO 12
GESTIONE DEI FILE 541

12.1 Introduzione 544
12.2 Organizzazione ed accesso a file 548
12.3 Le directory di file 556
12.4 Condivisione di file 559
12.5 Organizzazione di record a blocchi 562

12.6 Gestione della memoria secondaria 564
12.7 Gestione dei file in UNIX 572
12.8 Il file system di Windows NT 576
12.9 Sommario 581
12.10 Letture raccomandate 582
12.11 Problemi 582

PARTE SESTA
SISTEMI DISTRIBUITI

CAPITOLO 13
ELABORAZIONE DISTRIBUITA, CLIENT/SERVER E CLUSTER 589

13.1 Il bisogno di un'architettura di protocollo 590
13.2 L'architettura del protocollo TCP/IP 591
13.3 L'architettura di protocollo OSI 599
13.4 Elaborazione client/server 601
13.5 Scambio di messaggi distribuito 612
13.6 Chiamate di procedura remote 617
13.7 Cluster 620
13.8 Wolfpack di Windows NT 625
13.9 Solaris MC 627
13.10 Sommario 630
13.11 Letture raccomandate 631
13.12 Problemi 632

CAPITOLO 14
GESTIONE DEI PROCESSI DISTRIBUITI 635

14.1 Migrazione dei processi 635
14.2 Stati globali distribuiti 644
14.3 Mutua esclusione distribuita 649
14.4 Stallo distribuito 660
14.5 Sommario 672
14.6 Letture raccomandate 673
14.7 Problemi 674

PARTE SETTIMA
SICUREZZA

CAPITOLO 15
SICUREZZA 677

15.1 Minacce alla sicurezza 678
15.2 Protezione 684

15.3 Intrusi 688
15.4 Virus e relative minacce 703
15.5 Sistemi fidati 712
15.6 Sicurezza di rete 716
15.7 La sicurezza in Windows NT 723
15.8 Sommario 728
15.9 Letture raccomandate 730
15.10 Problemi 730
Appendice 15A Cifratura 732

APPENDICE A
ANALISI DELLE CODE 739
A.1 Perché l'analisi delle code? 740
A.2 Modelli di code 742
A.3 Code per singolo server 748
A.4 Code multiserver 751
A.5 Reti di code 752
A.6 Esempi 756
A.7 Altri modelli di coda 760
A.8 Letture raccomandate 760
Allegato A Quel tanto che basta di probabilità e statistica 761

APPENDICE B
PROGETTAZIONE ORIENTATA AGLI OGGETTI 767
B1 Motivazione 767
B2 Concetti object-oriented 768
B.3 Benefici della progettazione orientata agli oggetti. 772
B.4 CORBA 773
B.5 Letture raccomandate 776

APPENDICE C
PROGRAMMAZIONE E PROGETTAZIONE DI SISTEMI OPERATIVI 777
C.1 Progetti per insegnare sistemi operativi 777
C.2 Nachos 778
C.3 Progetti di programmazione 779
C.4 Assegnazione di letture e relazioni 780

APPENDICE D
OSP: UN AMBIENTE PER LA PROGETTAZIONE DI SISTEMI OPERATIVI 781
D.1 Panoramica 781
D.2 Aspetti innovativi di OSP 784
D.3 Confronto con altri software destinati alla didattica di sistemi operativi 786
D.4 La distribuzione del software di OSP 787

D.5 Mailing list di OSP 787
D.6 Progetti futuri 788

APPENDICE E
BACI: IL SISTEMA DI BEN ARI PER LA PROGRAMMAZIONE CONCORRENTE 789

E.1 Introduzione 789
E.2 BACI 790
E.3 Esempi di programmi BACI 793
E.4 Progetti in BACI 798
E.5 Miglioramenti del sistema BACI 801

GLOSSARIO 803

BIBLIOGRAFIA 813

INDICE ANALITICO 831

PREFAZIONE

Obiettivi

Questo libro tratta concetti, struttura e meccanismi dei sistemi operativi, e si propone di presentare nel modo più chiaro e completo possibile la natura e le caratteristiche dei sistemi più recenti.

Tale obiettivo rappresenta una sfida per diverse ragioni. In primo luogo, i sistemi di elaborazione per i quali si progettano sistemi operativi sono moltissimi e assai diversi tra loro: workstation a singolo utente e personal computer, sistemi condivisi di medie dimensioni, grandi mainframe, supercomputer e macchine specializzate, come i sistemi in tempo reale. La diversità non è in relazione soltanto con la capacità e la velocità delle macchine, ma risiede nelle applicazioni e nei requisiti di supporto del sistema. In secondo luogo, il rapido evolversi caratteristico dei sistemi di elaborazione non accenna a fermarsi, gran parte dei problemi chiave nella progettazione di un sistema operativo hanno una storia recente, e la ricerca in queste ed in nuove aree continua. Malgrado questa varietà e ritmo di cambiamento, determinati concetti fondamentali si applicano consistentemente ovunque, anche se in relazione allo sviluppo tecnologico del momento e alle particolari esigenze applicative. Questo testo intende fornire una trattazione completa dei fondamenti dell'architettura dei sistemi operativi, correlandoli alle attuali problematiche di progettazione e agli attuali sviluppi dei sistemi operativi.

Il lettore potrà acquisire una solida conoscenza dei meccanismi chiave dei sistemi operativi moderni, delle diverse alternative e decisioni inerenti la progettazione dei sistemi operativi e del contesto entro cui funziona il sistema (hardware, altri programmi di sistema, programmi applicativi, utenti interattivi).

I sistemi scelti come esempio

Questo testo si ripromette di fornire al lettore la conoscenza dell'architettura di base e dei problemi d'implementazione dei sistemi operativi contemporanei, e quindi una trattazione puramente concettuale o teorica sarebbe fuori luogo. Per illustrare i concetti ed esemplificare i prin-

cipi di fondo, collegandoli alle scelte progettuali da adottare in concreto, si è fatto riferimento a tre sistemi operativi:

- **Windows NT**: un sistema a singolo utente e multitasking per personal computer, workstation e server. Essendo un sistema operativo di nuova concezione, si avvale, in modo lineare ed efficace, di molti sviluppi recentissimi della tecnologia dei sistemi operativi. Windows NT è uno dei primi importanti sistemi operativi commerciali ad accogliere i principi della progettazione orientata agli oggetti.
- **UNIX**: un sistema operativo multiutente, concepito in origine per minicomputer, ma implementato su un'ampia gamma di macchine, dai potenti microcomputer ai supercomputer. La versione cui soprattutto si fa riferimento in questo libro è SVR4, dotata di numerose caratteristiche fra le più attuali dei sistemi operativi.
- **Solaris**: la versione commerciale di UNIX più diffusa. Solaris include il multithread e altre caratteristiche non presenti in SVR4 e nella maggior parte delle altre versioni di UNIX.

Questi tre sistemi sono stati scelti per la loro rilevanza e rappresentatività. Adottando la strategia usata dallo stesso autore in *Computer Organization and Architecture*, le osservazioni sui sistemi presi ad esempio compaiono ovunque nel testo, piuttosto che concentrate in un singolo capitolo o in un'appendice. Ad esempio, parlando della concorrenza, se ne descrivono i meccanismi in ciascuno dei tre sistemi modello e si esaminano le motivazioni delle relative scelte di progetto, valide per ciascun sistema. Questa strategia consente di evidenziare prontamente, con esempi concreti, i concetti esposti in un certo capitolo.

A chi è rivolto questo libro

Questo libro è rivolto ad un pubblico di studenti e di professionisti tecnici. Come libro di testo è stato formulato in vista di corsi universitari di sistemi operativi per Informatica, Ingegneria Informatica e Ingegneria Elettronica: sono qui trattati infatti tutti gli argomenti raccomandati da organizzazioni come IEEE ed ACM per i curricula universitari relativi alle tecnologie dell'informazione. Può essere utilizzato anche come testo di base per l'autoapprendimento.

Organizzazione del testo

Il libro è organizzato in sette parti.

I. **Elementi di base**: offre una panoramica dell'architettura e dell'organizzazione dei sistemi di elaborazione, con particolare riguardo agli argomenti correlati alla progettazione dei sistemi operativi; questa prima parte presenta in sintesi anche i contenuti degli altri capitoli.

II. **Processi**: analizza in dettaglio i processi, il multithread, il multiprocessing simmetrico (SMP) e i microkernel, e gli aspetti chiave della concorrenza in un singolo sistema, soffermandosi sui problemi della mutua esclusione e dello stallo.

III. **La memoria**: presenta un panorama complessivo delle tecniche di gestione della memoria.

IV. **Lo scheduling**: esamina e confronta i vari approcci alla schedulazione dei processi, la schedulazione dei thread, la schedulazione SMP e quella dei sistemi in tempo reale.

V. **Input/Output e file**: tratta i problemi relativi al controllo delle funzioni di I/O. Un'attenzione speciale è dedicata all'I/O del disco, fondamentale per le prestazioni del sistema, quindi si affronta in generale il problema della gestione dei file.

VI. **Sistemi distribuiti**: passa in rassegna le più significative linee di tendenza dell'elaborazione di rete, in particolare TCP/IP, l'elaborazione client/server e i cluster, e descrive anche alcune delle più rilevanti aree progettuali nello sviluppo dei sistemi operativi distribuiti.

VII. **Sicurezza**: esamina i tipi di rischio ed i meccanismi per la sicurezza dei computer e delle reti.

Un sommario più dettagliato del contenuto di ciascun capitolo si trova comunque alla fine del Capitolo 2. Completano il volume un vasto glossario, una lista degli acronimi di uso più frequente e una bibliografia. Ciascun capitolo è corredato di esercizi e di un elenco di letture integrative.

Servizi Internet per docenti e studenti

Esiste una pagina Web appositamente creata per accompagnare il libro nell'edizione originale, che può essere di aiuto a studenti e docenti. La pagina contiene link a siti interessanti, lucidi tratti dalle illustrazioni del libro in formato PDF (Adobe Acrobat), una serie di appunti per lezioni, che si possono usare come lucidi o come riassunti per studenti, nonché le indicazioni per iscriversi alla mailing list del libro. L'indirizzo Web è http://www.williamstallings.com. La mailing list è stata creata affinché coloro che usano questo libro possano scambiare informazioni, suggerimenti e domande fra di loro e con l'autore. Gli eventuali errori, di stampa o di altro genere, saranno elencati in una errata corrige disponibile allo stesso indirizzo.

Progetti di sistema operativo

Molti docenti reputano fondamentale che un corso di sistemi operativi comprenda anche un progetto, o un insieme di progetti, grazie ai quali lo studente possa acquisire esperienza pratica e consolidare le nozioni apprese dal testo. Per quanto riguarda le esercitazioni da proporre agli studenti, il supporto offerto da questo testo è senza confronti. Esso fornisce informazioni su tre

pacchetti software, utilizzabili per l'implementazione di progetti: OSP e NACHOS per sviluppare componenti di un sistema operativo, BACI per studiare i meccanismi della concorrenza. Il manuale per il docente contiene inoltre una serie di piccoli progetti di programmazione, sviluppabili in una settimana o due, inerenti un'ampia gamma di problematiche, che possono essere implementati in qualunque linguaggio e su qualsiasi piattaforma. Per maggiori dettagli, si consulti l'Appendice C.

Le novità della terza edizione

La seconda edizione di questo libro risale al 1995, e nel frattempo si sono prodotti continuamente progressi e innovazioni, che sono stati catturati al volo per questa nuova edizione, pur conservando il carattere di trattazione generale degli argomenti. Al fine di procedere ad un aggiornamento, la seconda edizione del libro è stata rivista da diversi docenti che l'avevano adottata come testo all'epoca. In molti passi del testo la parte discorsiva è stata chiarita e sintetizzata, le illustrazioni migliorate e, dopo averli testati, sono stati inseriti anche numerosi nuovi esercizi.

A parte queste modifiche, volte a migliorare l'efficacia didattica e ad agevolare gli studenti, sono state introdotte in tutto il manuale anche innovazioni sostanziali, di cui le principali sono:

- **Multithread**: la trattazione è più ampia e comprende l'analisi dei thread a livello utente e a livello kernel, la schedulazione e la migrazione dei thread. Anche la parte sul multithread in Windows NT è più vasta, ed è completamente nuova quella sul multithread in Solaris.
- **Multiprocessing simmetrico (SMP)**: il discorso è molto più completo e riguarda anche, in modo più dettagliato, la gestione di processi e la schedulazione dei processi e dei thread.
- **Microkernel**: la sezione sul microkernel è completamente nuova.
- **Cluster**: negli ultimi anni, i cluster sono diventati un'importante forma di organizzazione dei sistemi di elaborazione e dei sistemi operativi, perciò è stata inserita un'apposita sezione sull'argomento, assieme ad una trattazione sugli approcci di Windows NT Wolfpack e di Solaris MC.
- **Progettazione orientata agli oggetti**: anche sulla progettazione orientata agli oggetti il discorso è stato ampliato, in particolare arricchendolo con esempi tratti dai sistemi operativi di riferimento. In questa edizione, l'Appendice contiene un'introduzione a CORBA.
- **Maggior supporto al docente**: è già stato ricordato come questo testo fornisca un buon supporto per le esercitazioni; inoltre, è stato anche arricchito il sito Web dedicato al libro.

Ringraziamenti

Questa nuova edizione ha potuto avvalersi della revisione di un gruppo di persone che hanno offerto generosamente il loro tempo e la loro esperienza, proponendo molti utili suggerimenti.

Questi i loro nomi: John B. Connely (Cal Poly State Univ. San Louis Obispo), Michael P. Deignan, John Graham (Univ. of Delaware), Jeff Jirka, Wayne Madison (Clemson), Matt W. Mutka (Michigan State University), Bina Ramamurthy (SUNY - Buffalo), e Jaleh Rezaie (Easter Kentucky Univ.). Tracy Camp (Univ. of Alabama) ha revisionato il nuovo materiale sui microkernel. Ewan Grantham (Grantham & Associates), Bill Kuhn (Telecommunications Research Associates), Dave Porter, e Bill Potvin (STK/NGS/TerIS) hanno revisionato il materiale su Windows NT. Jerry Gulla (Sun Microsystems) e William Tepfenhart (AT&T) hanno revisionato il materiale su UNIX.

Vorrei anche ringraziare Michael Kifer e Scott A. Smolka (SUNY - Stony Brook) per aver contribuito all'Appendice D; Bill Bynum (College of William and Mary) e Tracy Camp (Univ. of Alabama) per aver contribuito all'Appendice E; Steve Taylor (Worcester Polytechnic Institute) per aver contribuito ai progetti di programmazione ed agli esercizi di lettura/relazione del manuale del docente.

l'Autore

PREFAZIONE ALL'EDIZIONE ITALIANA

In un settore in continua evoluzione come l'informatica, e sul tema dei sistemi operativi in particolare, si assiste ad un continuo inseguimento dell'editoria nei confronti delle "novità" immesse sul mercato dai produttori di software; a queste novità si affianca un insieme di testi, ormai diventati classici per il mondo universitario, sui principi fondamentali teorici alla base dei sistemi operativi.

Questo volume si presenta come un testo che riesce a coprire entrambi gli aspetti: contiene cioè un'esposizione chiara, ricca d'esempi ed esercizi, dei principi fondamentali, consentendo agli studenti delle discipline informatiche di comprendere l'argomento vastissimo dei sistemi operativi; allo stesso tempo vi si trova una raccolta d'informazioni, molto utili ed aggiornate, sulle tendenze piu recenti dello sviluppo di software, e sull'impatto che nuove architetture hardware potranno avere sui sistemi operativi del prossimo futuro.

Siamo di fronte a una consistente riorganizzazione delle Università italiane, con l'introduzione di titoli universitari di primo livello, di corsi post diploma superiore, di master industriali e di lauree specialistiche, che sostituiranno le tradizionali lauree delle Facoltà di Scienze ed Ingegneria, e renderanno il percorso di formazione iniziale più vicino a quello che si compie nel resto dell'Europa. Il libro di Stallings si presta a diversi percorsi didattici, secondo gli interessi del docente, il livello d'approfondimento e il "taglio" che intende seguire nel corso; ma cosa ancor più importante, a mio giudizio, non esaurisce la propria funzione con il superamento dell'esame universitario.

Ci troviamo in presenza di un testo utilissimo per gli approfondimenti individuali, da consultare frequentemente anche dopo la conclusione del percorso universitario, nel corso della vita professionale, e che il sistemista può tenere in considerazione per integrare e completare le proprie competenze professionali, in un percorso di formazione individuale permanente.

Gabriella Dodero
Docente di Sistemi Operativi per il Corso di Laurea in Informatica
Università di Genova

ELEMENTI DI BASE

Parte 1

Questa prima parte passa in rassegna gli elementi di base, prerequisiti per comprendere i temi più specifici affrontati nel libro, e presenta i concetti fondamentali della struttura interna dell'architettura dei sistemi di elaborazione e dei sistemi operativi.

Capitolo 1
Introduzione ai sistemi di elaborazione

Un sistema operativo funge da mediatore e da coordinatore fra programmi applicativi, programmi di utilità e utenti da un lato, e l'hardware del sistema di elaborazione dall'altro: infatti, per comprendere le funzionalità del sistema operativo e le problematiche suscitate dalla sua progettazione, è necessaria anche una certa conoscenza dell'organizzazione e dell'architettura dei sistemi di elaborazione. Il Capitolo 1 ricapitola brevemente cosa sono il processore, la memoria e gli elementi di I/O di un sistema di elaborazione.

Capitolo 2
Introduzione ai sistemi operativi

L'architettura dei sistemi operativi è un campo vastissimo, ed è facile, se ci si dedica ad un problema particolare, perdersi nei dettagli smarrendo la visione d'insieme. Il Capitolo 2 traccia una sintetica panoramica, cui il lettore potrà fare riferimento, nel corso della lettura, appunto per recuperare una visione globale del problema. Una volta definiti obiettivi e funzioni di un sistema operativo, sono descritti alcuni sistemi storicamente rilevanti; questa descrizione consente di individuare alcuni principi fondamentali di progettazione presentandoli in un ambiente semplice, in modo che le relazioni fra le varie funzioni dei sistemi operativi risultino ben chiare. In seguito, si mettono in luce alcune caratteristiche importanti dei sistemi operativi moderni. Nel corso del libro, man mano che si affrontano diversi argomenti, sono presentati sia i principi fondamentali consolidati, sia le innovazioni più recenti nell'architettura dei sistemi operativi: occorre, infatti, che il lettore sia consapevole dei più tradizionali e consolidati approcci alla progettazione, come di quelli più

innovativi. Il capitolo si conclude con una panoramica generale sull'architettura di Windows NT e UNIX, cui fare riferimento anche in seguito, quando l'analisi diverrà più dettagliata.

CAPITOLO 1

INTRODUZIONE AI SISTEMI DI ELABORAZIONE

Un sistema operativo sfrutta le risorse hardware di uno o più processori per fornire un insieme di servizi agli utenti e, inoltre, gestisce la memoria e i dispositivi di I/O (*ingresso uscita*). Di conseguenza, prima di esaminare i sistemi operativi, è fondamentale disporre di alcune nozioni di base sul funzionamento dell'hardware del sistema.

Questo capitolo fornisce un'introduzione all'hardware dei sistemi di elaborazione. Nella maggior parte dei casi, la trattazione sarà breve, poiché si presume che il lettore abbia familiarità con l'argomento, ma alcuni punti verranno esaminati più in dettaglio, a causa della loro importanza per gli argomenti analizzati in seguito.

1.1 Elementi di base

Al livello più astratto, un computer consiste di processore, memoria e componenti di I/O, con uno o più moduli per ciascun tipo. Queste componenti sono interconnesse nel modo appropriato per svolgere la funzione principale del computer, che consiste nell'eseguire programmi. Gli elementi strutturali più importanti sono quattro:

- **Processore**: controlla le operazioni del computer e svolge le sue funzioni di elaborazione dei dati. Quando il processore è unico, spesso viene definito CPU (*unità di elaborazione centrale*, Central Processing Unit).

- **Memoria principale**: memorizza dati e programmi. Questa memoria è tipicamente volatile, e viene anche definita memoria primaria.
- **Moduli di I/O**: trasferiscono i dati fra il computer e il suo ambiente esterno, variamente configurato (dispositivi di memoria secondaria, dispositivi per la comunicazione, terminali.
- **Interconnessione del sistema**: strutture e meccanismi che provvedono alla comunicazione fra processori, memoria centrale e moduli di I/O.

La Figura 1.1 illustra queste componenti di livello più astratto. Il processore svolge tipicamente funzioni di comando e controllo, una delle quali consiste nello scambiare dati con la memoria. A questo fine, il processore utilizza due registri interni: MAR (*registro degli indirizzi di memoria*, Memory Address Register), che specifica l'indirizzo in memoria per la successiva operazione di lettura o scrittura, e MBR (*registro buffer di memoria*, Memory Buffer Register), che contiene i dati da scrivere in memoria o riceve i dati letti dalla memoria. Similmente, un

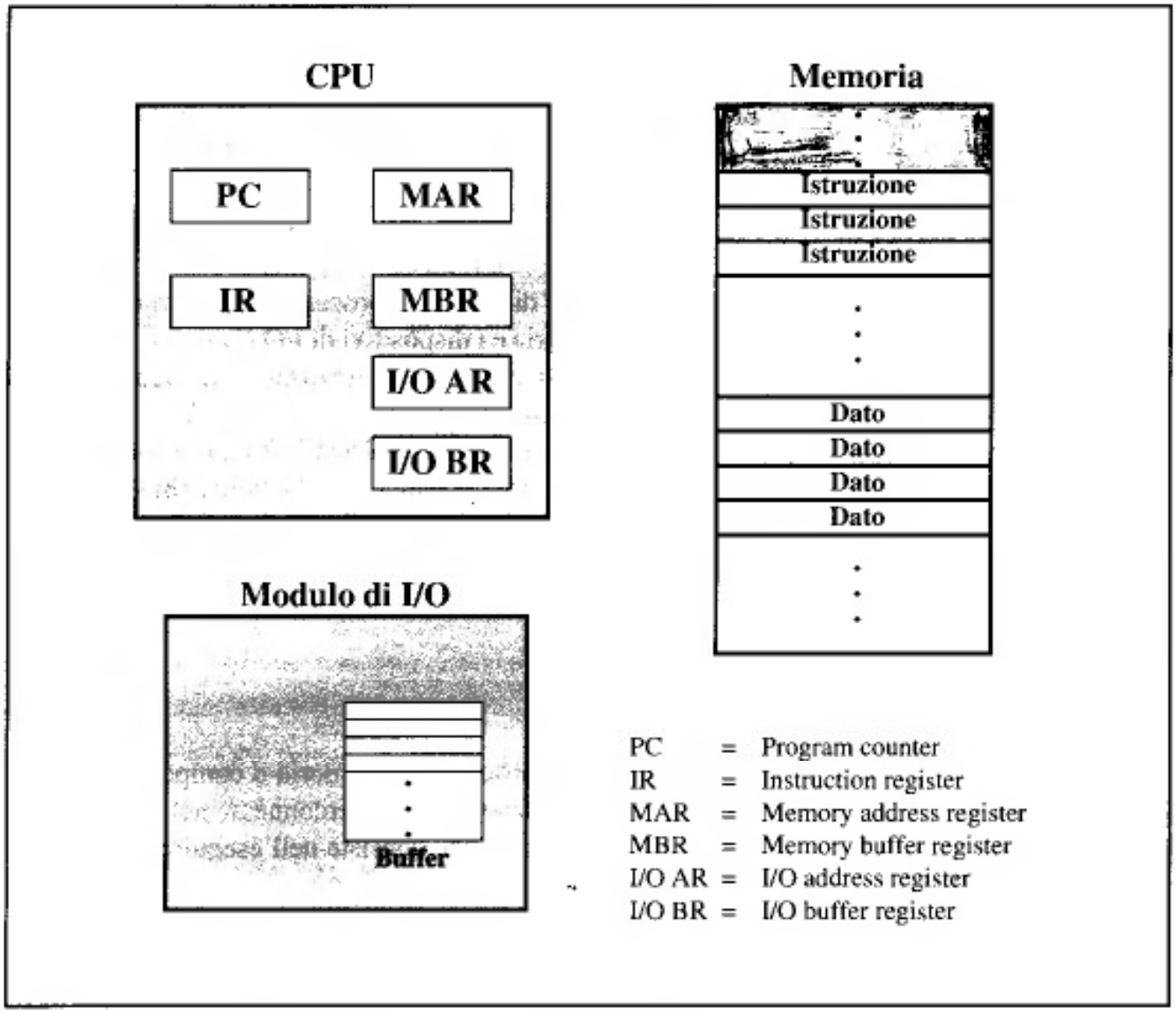


Figura 1.1 *Componenti di un computer: visione di alto livello*

registro I/O AR (*registro per gli indirizzi di ingresso uscita*, Input Output Address Register) specifica un particolare dispositivo di I/O, un registro I/O BR (*registro buffer ingresso uscita*, Input Output Buffer Register) è usato per scambiare i dati fra il modulo di I/O e il processore.

Un modulo di memoria consiste di un insieme di locazioni, definite da indirizzi numerati sequenzialmente, ciascuna delle quali contiene un numero binario, interpretabile o come istruzione o come dato. Un modulo di I/O trasferisce i dati dai dispositivi esterni al processore e alla memoria, e viceversa, e contiene buffer (*area di memoria temporanea*) interni per conservare temporaneamente i dati finché possono essere inviati.

1.2 Registri del processore

Un processore contiene un insieme di registri, che forniscono un livello di memoria più veloce e ridotto rispetto alla memoria principale e svolgono due funzioni:

- **Registri visibili all'utente**: permettono al programmatore a livello di linguaggio macchina o Assembler di minimizzare i riferimenti alla memoria principale, ottimizzando l'uso dei registri. Per quanto riguarda i linguaggi di alto livello, un compilatore ottimizzante cercherà di compiere scelte intelligenti riguardo alle variabili da assegnare ai registri e alle locazioni della memoria principale. In alcuni linguaggi di alto livello, come il C, il programmatore può suggerire al compilatore quali variabili dovrebbero essere memorizzate nei registri.
- **Registri di stato e di controllo**: sono usati dal processore per controllare le sue operazioni e dalle routine privilegiate del sistema operativo per controllare l'esecuzione dei programmi.

Non esiste una chiara separazione dei registri in queste due categorie: ad esempio, su alcune macchine il program counter (*contatore di programma*) è visibile all'utente, ma in molte altre ciò non accade. Ai fini della trattazione che segue, risulta però utile mantenere tale separazione.

Registri visibili all'utente

Si può fare riferimento ai registri visibili all'utente nel linguaggio macchina eseguito dal processore: tali registri sono generalmente disponibili per tutti i programmi, sia applicativi sia di sistema. I registri tipicamente disponibili sono quelli dei dati, degli indirizzi e dei condition code (*codici di condizione*).

Il programmatore può destinare i *registri dei dati* ad una molteplicità di funzioni. In alcuni casi, essi sono di tipo generale (*general purpose*) e possono essere usati in qualsiasi istruzione macchina che esegua operazioni sui dati; spesso però esistono delle restrizioni, come nel caso dei registri dedicati a operazioni in virgola mobile.

I *registri degli indirizzi* contengono gli indirizzi dei dati e delle istruzioni contenuti nella memoria principale; in alcuni casi, contengono una porzione dell'indirizzo usato nel calcolo dell'indirizzo completo. Questi registri possono essere anch'essi di tipo generale, oppure dedicati a un particolare modo di indirizzamento.

Elenchiamo alcuni esempi:

- **Index register** (*registro indice*): l'indirizzamento indicizzato è un modo comune di indirizzamento, che consiste nell'aggiungere un indice a un valore base per ottenere l'indirizzo effettivo.
- **Segment pointer** (*registro puntatore al segmento*): con l'indirizzamento segmentato la memoria è divisa in blocchi di lunghezza variabile. Un riferimento alla memoria consiste di un riferimento a un suo segmento particolare e di un offset entro il segmento stesso; in questo genere di indirizzamento (importante quando tratteremo la gestione della memoria nel Capitolo 7) viene usato un registro per memorizzare l'indirizzo base (locazione di partenza) del segmento. Ci possono essere registri multipli; ad esempio, uno per il sistema operativo (quando il codice del sistema operativo è in esecuzione nel processore) e uno per l'applicazione al momento in esecuzione.
- **Stack pointer** (*registro puntatore allo stack*): se è presente l'indirizzamento tramite stack (*pila*) visibile all'utente, allora lo stack è nella memoria principale ed esiste un registro dedicato che punta alla sua cima. Questo permette l'uso di istruzioni che non contengono campi indirizzo, come push e pop (per una discussione sulla gestione dello stack, vedi l'Appendice 1B).

Un ultimo gruppo di registri, almeno parzialmente visibili all'utente, memorizza i condition code, detti anche *flag*, che sono bit settati dall'hardware del processore, in seguito al risultato di determinate operazioni. Ad esempio, un'operazione aritmetica può produrre un risultato positivo o negativo, uno zero o un overflow; in aggiunta al risultato stesso, memorizzato in un registro o in memoria, viene settato anche un flag, che può poi comparire in un test per un'operazione di salto condizionato.

I bit dei condition code sono raccolti in uno o più registri. Di solito, costituiscono parte di un registro di controllo, e le istruzioni macchina permettono la lettura di questi bit per riferimento implicito, ma non lasciano che siano alterati dal programmatore.

In alcune macchine, una chiamata ad una procedura o ad una subroutine produce un salvataggio automatico di tutti i registri visibili all'utente, che vengono ricaricati alla fine della procedura stessa. Il salvataggio e il successivo ripristino dei registri sono effettuati dal processore, come parte delle istruzioni di chiamata e ritorno dalla subroutine e ciò permette a ciascuna procedura di usare questi registri indipendentemente dalle altre. Su altre macchine è responsabilità del programmatore salvare il contenuto dei registri visibili all'utente, prima di effettuare una chiamata di procedura, includendo le necessarie istruzioni nel programma. Pertanto, le funzioni di salvataggio e ripristino possono essere svolte a livello hardware o software, secondo il tipo di macchina.

Registri di controllo e di stato

Per controllare le operazioni del processore, vengono impiegati diversi registri, molti dei quali, nella maggior parte dei computer, non sono visibili all'utente. Ad alcuni è possibile accedere tramite istruzioni macchina eseguite in modalità di controllo o di sistema operativo.

Naturalmente, macchine diverse avranno organizzazioni dei registri diverse e li nomineranno diversamente. Presentiamo una lista abbastanza completa di tipi di registri, con una breve descrizione. In aggiunta ai MAR, MBR, I/O AR e I/O BR prima citati, per eseguire le istruzioni sono indispensabili anche i seguenti registri:

- **Program counter (PC)** (*contatore di programma*): contiene l'indirizzo della successiva istruzione da prelevare.
- **Instruction register (IR)** (*registro delle istruzioni*): contiene l'istruzione prelevata più recentemente.

L'architettura di tutti i processori include anche un registro, o un insieme di registri, comunemente noti come PSW (*parola di stato del programma*, Program Status Word), che contiene informazioni di stato: tipicamente, *flag* e altre informazioni di stato, come il bit per l'abilitazione/disabilitazione degli interrupt (*interruzione*) e il bit per la selezione del modo supervisore/utente.

In macchine che usano molti tipi di interruzioni, può essere presente un insieme di registri di interrupt con un puntatore a ciascuna routine per la *gestione dell'interrupt*. Se viene usato uno stack per implementare certe funzioni (ad esempio, le chiamate a procedura), allora è necessario un puntatore allo stack di sistema (vedi l'Appendice 1B). L'hardware per la gestione della memoria, trattato nel Capitolo 7, richiederà registri dedicati. Infine, si possono usare dei registri per il controllo delle operazioni di I/O.

Nel progettare l'organizzazione dei registri di controllo e di stato occorre tener conto di diversi fattori. Un fattore chiave è il supporto del sistema operativo, in quanto certi tipi d'informazione di controllo sono di utilità specifica per il sistema operativo. Se il progettista del processore conosce le funzioni del sistema operativo che verrà utilizzato può, entro certi limiti, adattare l'organizzazione dei registri al sistema stesso.

Un'altra scelta chiave è come distribuire l'informazione di controllo tra i registri e la memoria. Di norma, le prime centinaia o migliaia di parole di memoria sono dedicate a finalità di controllo: è compito del progettista decidere quanta informazione di controllo debba risiedere nei registri, più costosi e veloci, e quanta nella memoria principale, meno costosa e più lenta.

1.3 Esecuzione delle istruzioni

La funzione fondamentale di un computer consiste nell'eseguire un programma, cioè un insieme di istruzioni immagazzinate in memoria; questo lavoro è svolto dal processore che esegue le istruzioni specificate nel programma. Questa sezione introduce gli elementi chiave dell'esecuzione di un programma. Nella forma più semplice, eseguire un'istruzione consiste di due passi: il processore effettua il fetch (*prelievo*) delle istruzioni dalla memoria e le esegue una per volta. Eseguire un programma consiste nel ripetere il processo di fetch e di esecuzione dell'istruzione; l'esecuzione di un'istruzione può consistere di diverse operazioni e dipende dalla natura dell'istruzione stessa.

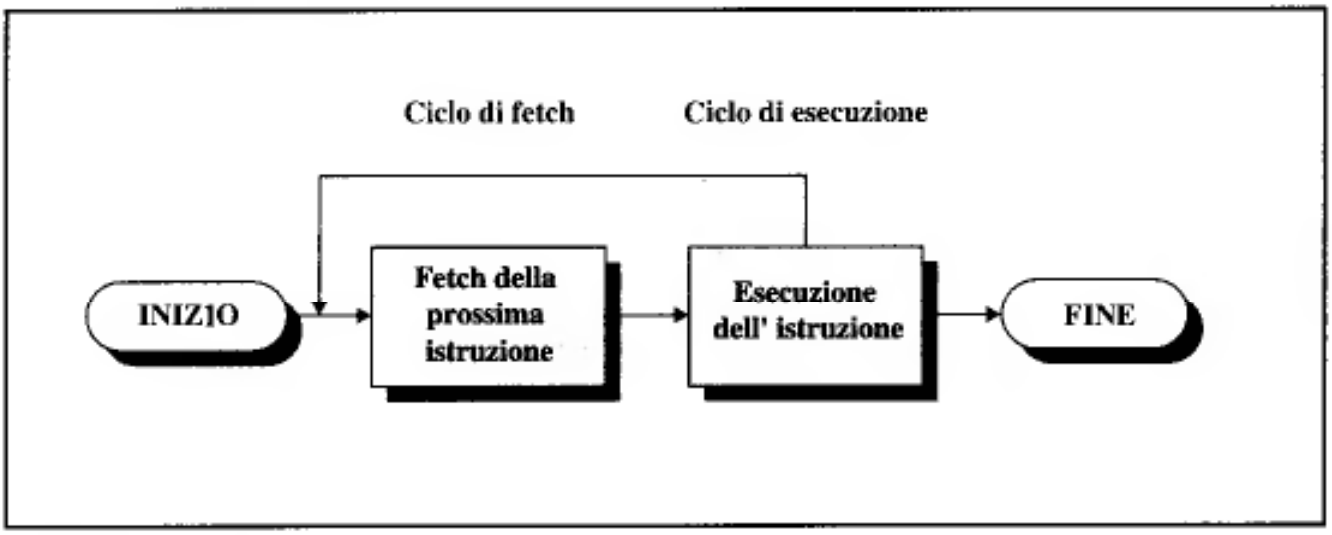


Figura 1.2 *Ciclo di base di una istruzione*

L'elaborazione richiesta per una singola istruzione è chiamata *ciclo di istruzione*. La Figura 1.2 illustra il ciclo di istruzione nella forma semplificata descritta sopra: i due passi sono definiti come *ciclo di fetch* e *ciclo di esecuzione*. L'esecuzione del programma termina solo se la macchina viene spenta, se si verifica un qualche errore impossibile da correggere, oppure in presenza di un'istruzione di arresto (*halt*).

Fetch ed esecuzione di un'istruzione

All'inizio di ciascun ciclo di istruzione, il processore preleva un'istruzione dalla memoria; in un processore tipico, il program counter (PC) memorizza l'indirizzo dell'istruzione successiva da prelevare. A meno di diverse indicazioni, dopo ciascun ciclo di fetch il processore incrementa sempre il PC, in modo che esso prelevi l'istruzione successiva in sequenza (cioè l'istruzione che si trova all'indirizzo di memoria immediatamente successivo). Prendiamo come esempio un computer in cui ciascuna istruzione occupi una parola di memoria a 16 bit, e il cui program counter contenga l'indirizzo di memoria 300: il processore preleverà l'istruzione alla locazione 300 e, nei successivi cicli di istruzione, preleverà le istruzioni alle locazioni 301, 302, 303 ecc. Questa sequenza può però essere cambiata nel modo che andiamo a spiegare.

L'istruzione prelevata è caricata in un registro del processore, noto come instruction register (IR). L'istruzione contiene bit che specificano l'azione che il processore deve effettuare; il processore interpreta l'istruzione ed esegue l'azione richiesta. In generale, queste azioni si raggruppano in quattro categorie:

- **Processore-memoria:** i dati possono essere trasferiti dal processore alla memoria o viceversa.
- **Processore-I/O:** i dati possono essere trasferiti da un dispositivo periferico al processore o viceversa, attraverso i moduli di I/O.
- **Elaborazione dei dati:** il processore può effettuare alcune operazioni, aritmetiche o logiche, sui dati.

- **Controllo:** un'istruzione può modificare la sequenza di esecuzione. Ad esempio, il processore può prelevare un'istruzione dalla locazione 149, la quale specifica che l'istruzione successiva è nella locazione 182. Il processore ricorderà questo fatto, caricando nel program counter 182, e, al successivo ciclo di fetch, l'istruzione sarà prelevata dalla locazione 182 invece che dalla 150.

L'esecuzione di un'istruzione può anche combinare diverse azioni tra quelle ora descritte.

Consideriamo un semplice esempio, con una macchina ipotetica dotata delle caratteristiche elencate nella Figura 1.3. Il processore contiene un singolo registro per i dati, chiamato accumulatore (AC). Sia le istruzioni sia i dati sono lunghi 16 bit, quindi conviene organizzare la memoria usando locazioni a 16 bit, dette parole. Il formato dell'istruzione prevede quattro bit per gli opcode (*codice operativo*), quindi ci possono essere fino a $2^4 = 16$ differenti opcode, ed è possibile indirizzare direttamente fino a $2^{12} = 4096$ (4K) parole di memoria.

La Figura 1.4 presenta l'esecuzione parziale di un programma illustrando le porzioni di memoria pertinenti e i registri del processore. Questo frammento di programma aggiunge il conte-

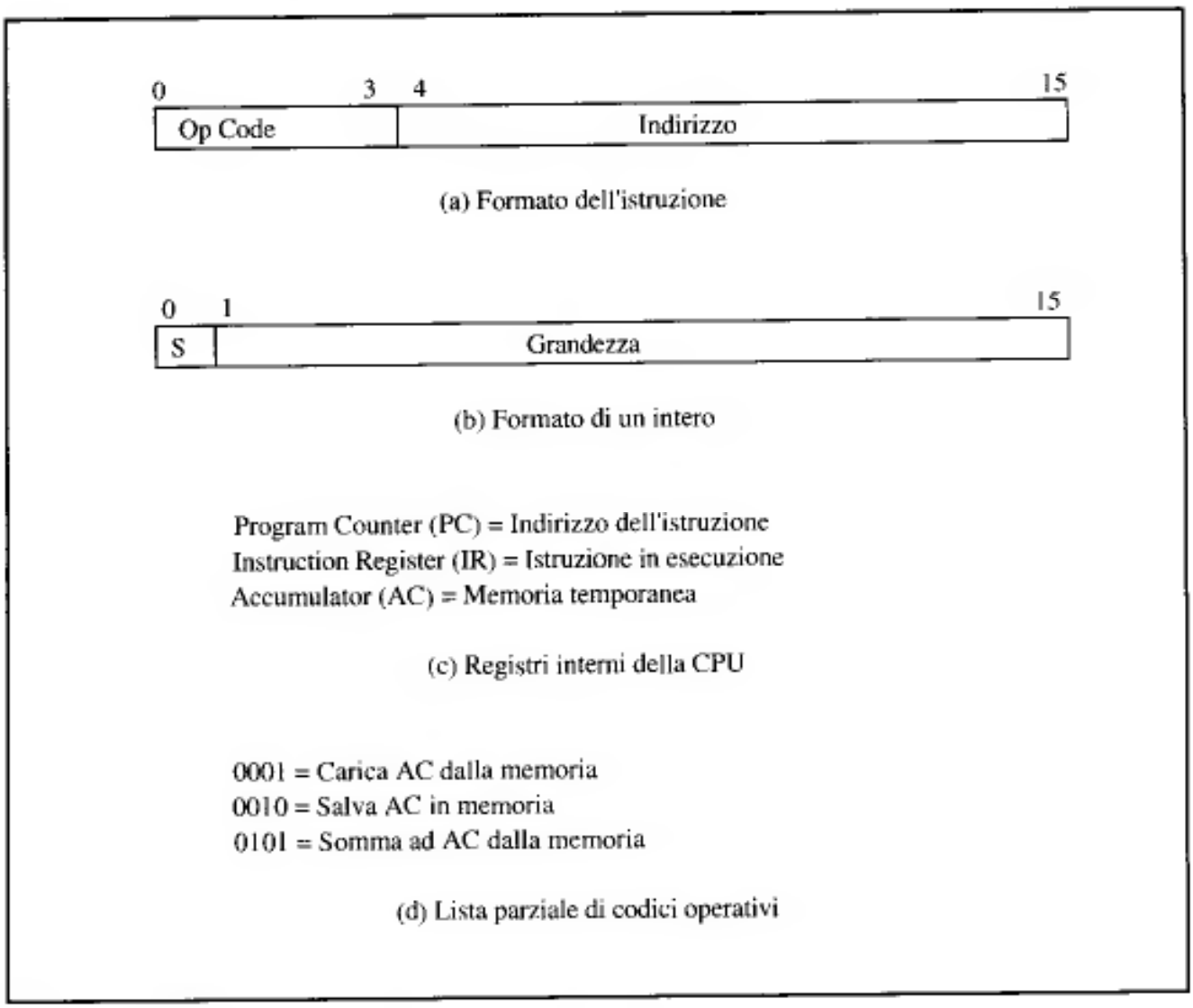


Figura 1.3 *Caratteristiche di una macchina ipotetica*

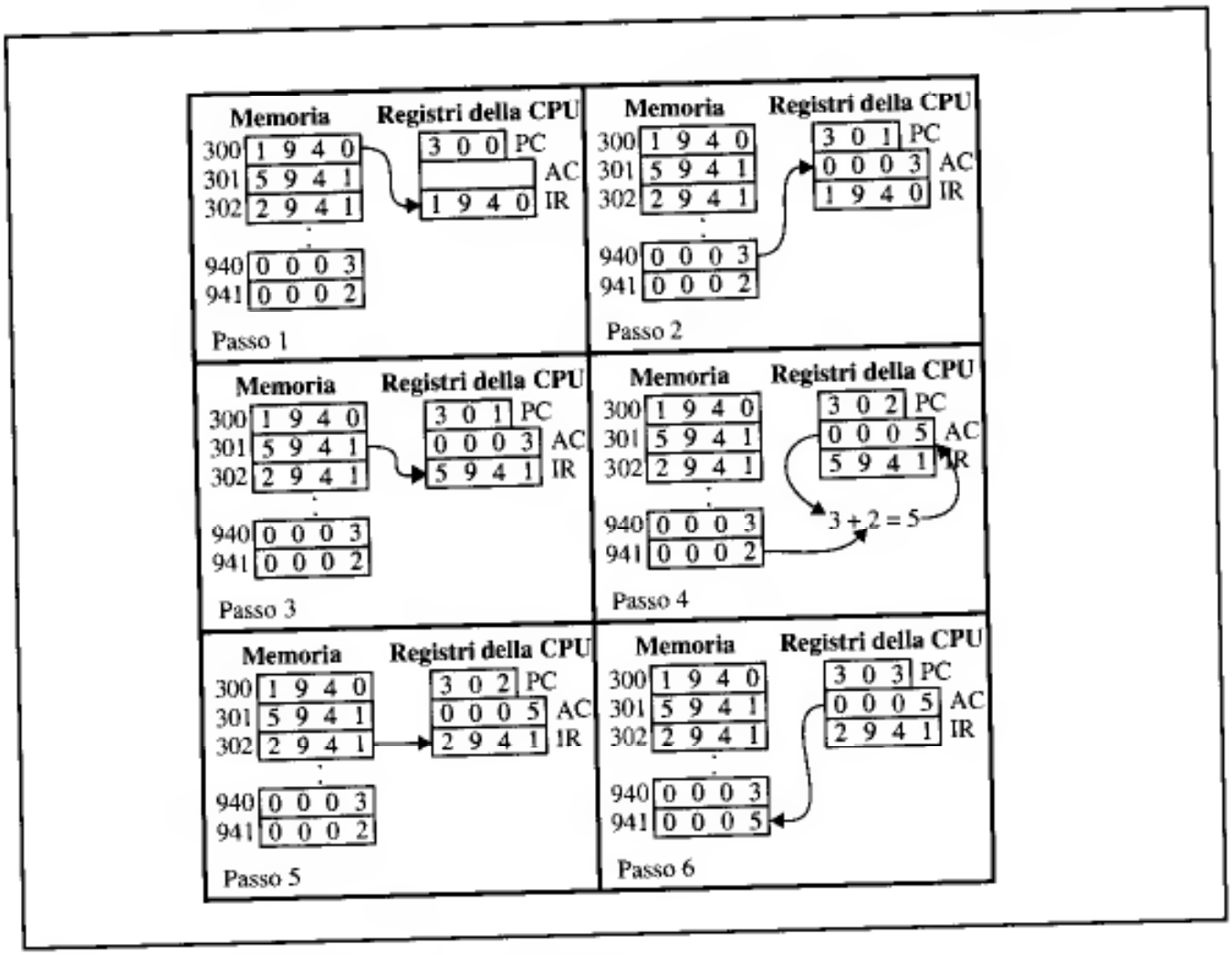


Figura 1.4 *Esempio di esecuzione di un programma*

nuto della parola di memoria all'indirizzo 940 al contenuto di quella all'indirizzo 941, dove memorizza il risultato. Sono richieste tre istruzioni, descrivibili come tre fetch e tre cicli di esecuzioni.

1. Il PC contiene 300, l'indirizzo della prima istruzione, che viene caricata nell'instruction register (IR). È da notare che questo processo comporterebbe l'uso del registro degli indirizzi di memoria (MAR) e del registro buffer di memoria (MBR) ma, per semplificare, ignoriamo questi registri intermedi.
2. I primi quattro bit in IR indicano che AC dev'essere caricato con una parola di memoria, e i rimanenti 12 bit ne specificano l'indirizzo, che è 940; il PC viene incrementato.
3. Viene prelevata l'istruzione successiva.
4. Il vecchio valore contenuto in AC e quello della locazione 941 sono sommati e il risultato è immagazzinato in AC; il PC viene incrementato.
5. Viene prelevata l'istruzione successiva.
6. Il contenuto di AC è salvato nella locazione 941; il PC viene incrementato.

In questo esempio, per sommare il contenuto della locazione 940 a quello della 941 sono necessari tre cicli di istruzione, ognuno costituito da un ciclo di fetch e uno di esecuzione ma, con un set di istruzioni più complesso, sarebbero necessari meno cicli. La maggior parte dei processori moderni ha istruzioni che contengono più di un indirizzo, quindi il ciclo di esecuzione per una determinata istruzione può comprendere più di un riferimento alla memoria. Inoltre, invece di riferirsi alla memoria, l'istruzione può specificare un'operazione di I/O.

Funzioni di I/O

Abbiamo finora descritto le operazioni del computer controllate dal processore, in particolare l'interazione fra il processore e la memoria, accennando appena al ruolo dei componenti di I/O.

Un modulo di I/O, ad esempio un controller (*dispositivo di controllo*) del disco, può scambiare dati direttamente col processore. Il processore può iniziare una lettura o scrittura in memoria, indicando l'indirizzo di una locazione specifica, e, allo stesso modo, può anche leggere o scrivere dati in un modulo di I/O; in quest'ultimo caso, il processore identifica un dispositivo specifico, controllato da un particolare modulo di I/O. Perciò, si potrebbe avere una sequenza di istruzioni, simile nella forma a quella della Figura 1.4, con istruzioni di I/O al posto di quelle che si riferiscono alla memoria.

Talvolta, è preferibile che i trasferimenti di I/O avvengano direttamente in memoria; in tal caso il processore concede al modulo di I/O il permesso di leggere o scrivere in memoria, in modo che il trasferimento I/O-memoria avvenga senza impegnare il processore. Durante tale trasferimento il modulo di I/O esegue comandi di lettura o scrittura in memoria, sollevando il processore dalla responsabilità dello scambio. L'operazione è nota come direct memory access (*accesso diretto alla memoria*, DMA), e verrà esaminata più avanti nel corso di questo capitolo.

1.4 Interruzioni

Praticamente, tutti i computer funzionano in modo che altri moduli (I/O, memoria) possano interrompere l'elaborazione normale del processore. La Tabella 1.1 elenca le più comuni classi di interruzioni (*interrupt*).

La funzione principale delle interruzioni è migliorare l'efficienza dell'elaborazione: infatti, la maggior parte dei dispositivi esterni è molto più lenta del processore. Supponiamo che il processore stia trasferendo dati ad una stampante, in base allo schema del ciclo di istruzioni della Figura 1.2: dopo ciascuna operazione di scrittura, il processore deve mettersi in pausa e rimanere inattivo finché la stampante non abbia concluso e la lunghezza di questa pausa può essere dell'ordine di molte centinaia o migliaia di cicli di istruzione, che non riguardano la memoria. Chiaramente, questo impiego del processore risulterebbe molto dispendioso.

La Figura 1.5a illustra questa situazione. Il programma utente esegue una serie di chiamate WRITE (*scrittura*), inserite nell'elaborazione; i segmenti di codice 1, 2 e 3 si riferiscono a sequenze di istruzioni che non comprendono operazioni di I/O. Le chiamate WRITE si riferiscono ad un programma di I/O che è una utilità di sistema, la quale effettua l'operazione di I/O vera

Tabella 1.1 *Classi di interruzioni*

Programma	Generate da alcune condizioni che possono verificarsi come risultato dell'esecuzione di un'istruzione, come l'overflow aritmetico, la divisione per zero, il tentativo di eseguire un'istruzione macchina illegale, o una violazione dello spazio di memoria permesso all'utente.
Timer	Generate da un timer all'interno del processore. Permettono al sistema operativo di effettuare determinate operazioni a intervalli di tempo regolari.
I/O	Generate da un controller di I/O per segnalare sia il completamento normale di un'operazione, sia situazioni di errore.
Errori hardware	Generate da problemi hardware, come una caduta di tensione, o un errore di parità della memoria

e propria. Il programma di I/O consiste di tre sezioni:

- Una sequenza di istruzioni, etichettata 4 nella Figura, per preparare l'operazione di I/O. Essa può prevedere la copiatura dei dati, da inviare in uscita, in un buffer speciale, e la preparazione di parametri per un comando del dispositivo.
- Il comando di I/O vero e proprio. Non usando le interruzioni, una volta che questo comando è partito, il programma deve aspettare che il dispositivo di I/O completi la funzione richiesta. Durante l'attesa, esso effettua ripetutamente un'operazione di test per determinare se l'operazione di I/O sia finita.
- Una sequenza di istruzioni, etichettata 5 nella figura, per completare le operazioni. Può comprendere la gestione di un flag che indica il successo o il fallimento dell'operazione.

Poiché l'operazione di I/O può richiedere un tempo relativamente lungo, il programma di I/O è interrotto in attesa del completamento delle operazioni; pertanto, il programma utente resta bloccato nel punto della chiamata WRITE per un periodo di tempo considerevole.

Interruzioni e ciclo di istruzione

Con le interruzioni, il processore può essere utilizzato per eseguire altre istruzioni, durante un'operazione di I/O. Prendiamo in esame il flusso di esecuzione nella Figura 1.5b. Come nel caso precedente, il programma utente raggiunge un punto in cui effettua una chiamata di sistema sotto forma di chiamata WRITE. Il programma di I/O invocato in questo caso consiste solo del codice di preparazione e del comando di I/O vero e proprio; eseguite queste poche istruzioni, il controllo ritorna al programma utente. Nel frattempo, il dispositivo esterno è impegnato ad acquisire dati dalla memoria del computer ed a stamparli, operazione condotta contemporaneamente all'esecuzione delle istruzioni del programma utente.

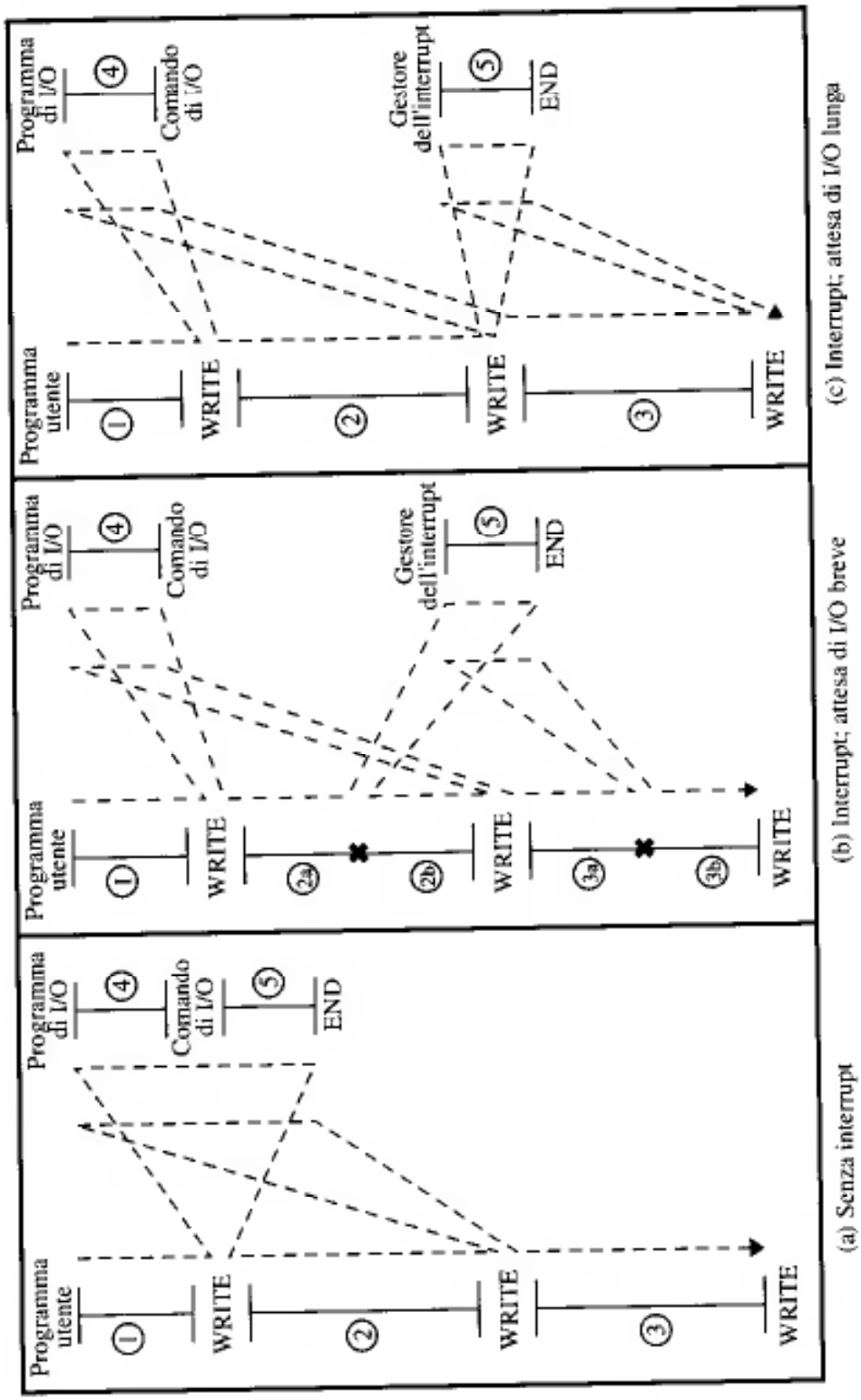


Figura 1.5 *Flusso di controllo del programma con e senza interrupt*

Quando il dispositivo esterno è pronto ad accettare altri dati, il relativo modulo di I/O manda al processore un *segnale di richiesta di interruzione*; il processore risponde sospendendo il programma corrente e saltando ad un programma di servizio per quel particolare dispositivo di I/O, noto come interrupt handler (*gestore di interruzione*). Infine, dopo che il dispositivo è stato servito, riprende ad eseguire il programma originale. I punti nei quali avvengono queste interruzioni sono indicati con **x** nella Figura 1.5b.

Dal punto di vista del programma utente, un interrupt è solo un'interruzione della normale sequenza di esecuzione: completata l'elaborazione dell'interrupt, l'esecuzione riprende (Figura 1.6). Pertanto, il programma utente non deve contenere alcun codice speciale per gestire le interruzioni: processore e sistema operativo sono responsabili della sua sospensione e poi della ripresa della sua esecuzione dal punto in cui era stata interrotta.

Per gestire gli interrupt, al ciclo dell'istruzione viene aggiunto *il ciclo di interruzione* (vedi Figura 1.7), dove il processore controlla se sia avvenuta un'interruzione, indicata dalla presenza dell'apposito segnale. Se non ci sono interruzioni in attesa, il processore passa al ciclo di fetch e preleva l'istruzione successiva del programma corrente; se c'è un'interruzione in attesa, il processore sospende l'esecuzione del programma corrente ed esegue una routine per la *gestione dell'interruzione*. Il programma di gestione degli interrupt in genere è parte del sistema operativo. Tipicamente, questo programma determina la natura dell'interruzione ed esegue le azioni necessarie. Ad esempio, nel nostro caso, il gestore di interrupt determina quale modulo di I/O abbia generato l'interruzione, e può saltare ad un programma che scriverà ulteriori dati in quel modulo di I/O. Quando la routine di gestione dell'interruzione è completata, il processore può riprendere ad eseguire il programma utente dal punto in cui era stato interrotto.

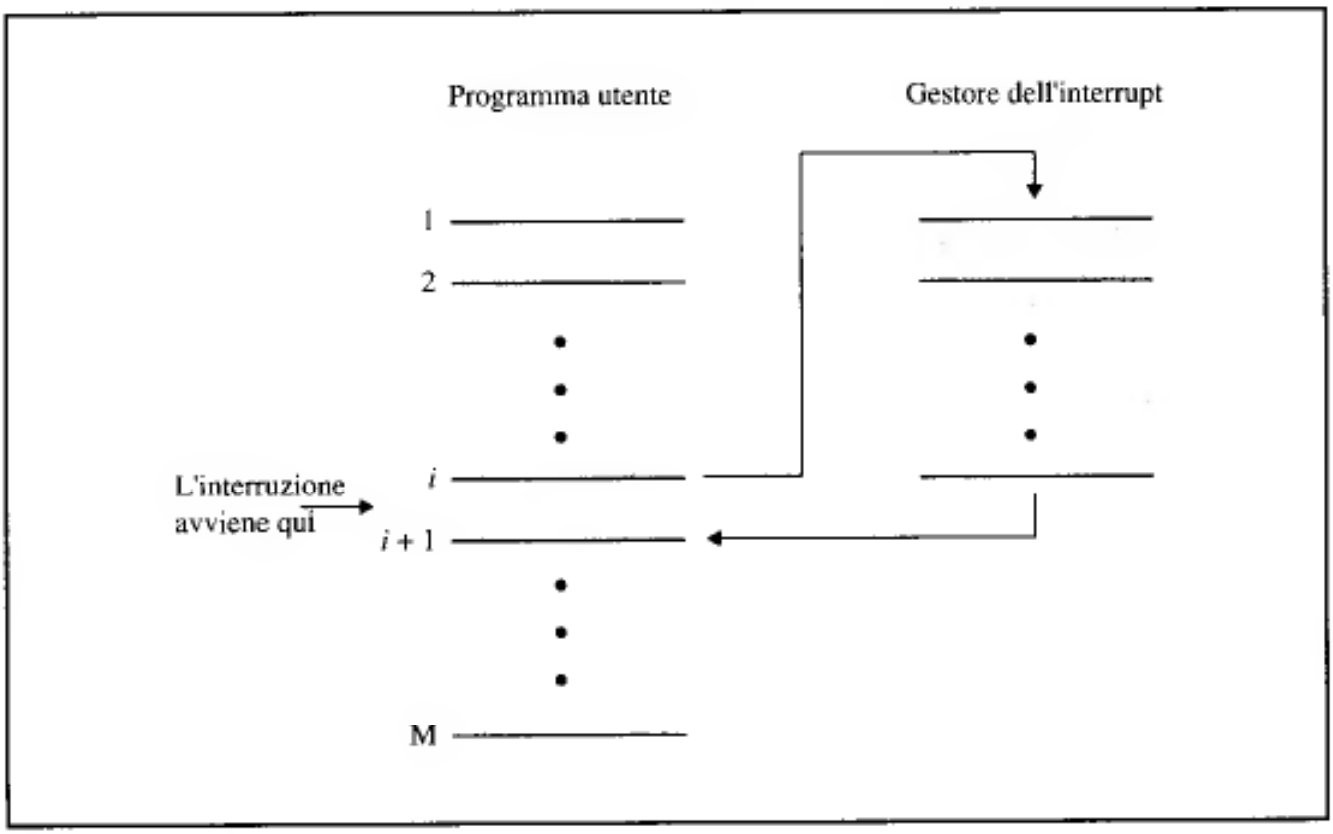


Figura 1.6 *Trasferimento del controllo attraverso interruzioni*

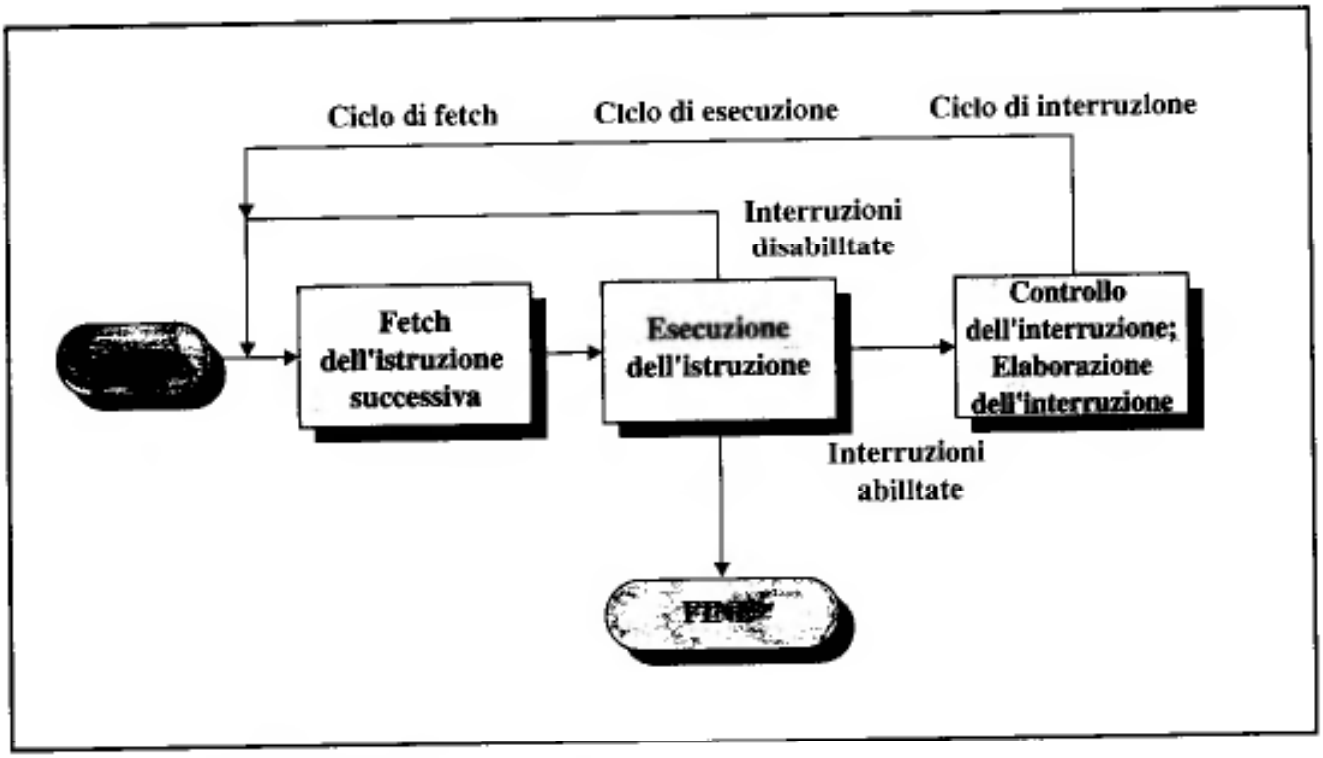


Figura 1.7 *Ciclo di istruzione con interruzioni*

Chiaramente, questo processo comporta un certo overhead. Per determinare la natura dell'interrupt e per decidere le azioni appropriate, nel gestore dell'interruzione occorre eseguire un certo numero di istruzioni in più. Tuttavia, poiché in attesa di un'operazione di I/O andrebbe perduta una quantità di tempo relativamente grande, è possibile utilizzare il processore con maggior efficienza, appunto servendosi delle interruzioni.

Per stimare il guadagno in efficienza, esaminiamo la Figura 1.8, che rappresenta un diagramma dei tempi basato sul flusso di controllo delle Figure 1.5a e 1.5b. Nelle Figure 1.5b e 1.8 si assume che il tempo richiesto per le operazioni di I/O sia relativamente breve, inferiore al tempo necessario per completare l'esecuzione delle istruzioni comprese fra due operazioni di scrittura nel programma utente. Il caso tipico, specialmente per un dispositivo lento come una stampante, è che l'operazione di I/O richieda molto più tempo rispetto all'esecuzione di una sequenza di istruzioni utente. La Figura 1.5c illustra questo caso: il programma utente raggiunge la seconda chiamata WRITE prima che l'operazione di I/O generata dalla precedente chiamata sia completata, perciò il programma utente rimane bloccato in quel punto. Completata la precedente operazione di I/O, è possibile eseguire la nuova chiamata WRITE, e può partire una nuova operazione di I/O. La Figura 1.9 illustra le sequenze dei tempi di esecuzione, con e senza l'uso delle interruzioni. Come si nota, si guadagna anche questa volta in efficienza: una parte del tempo di esecuzione dell'operazione di I/O si sovrappone all'esecuzione delle istruzioni utente.

Elaborazione delle interruzioni

Il verificarsi di un'interruzione genera una serie di eventi (tanto a livello dell'hardware che del software), illustrati nella Figura 1.10. Quando un dispositivo completa un'operazione di I/O, si produce questa sequenza di eventi:

1. Il dispositivo invia un segnale di interrupt al processore.
2. Il processore, prima di rispondere alla richiesta, termina di eseguire l'istruzione corrente, come indicato nella Figura 1.7.
3. Il processore controlla che si sia verificata un'interruzione e manda un segnale di acknowledgment (*conferma*) al dispositivo che l'ha generata. L'acknowledgment permette al dispositivo di rimuovere il suo segnale di interrupt.

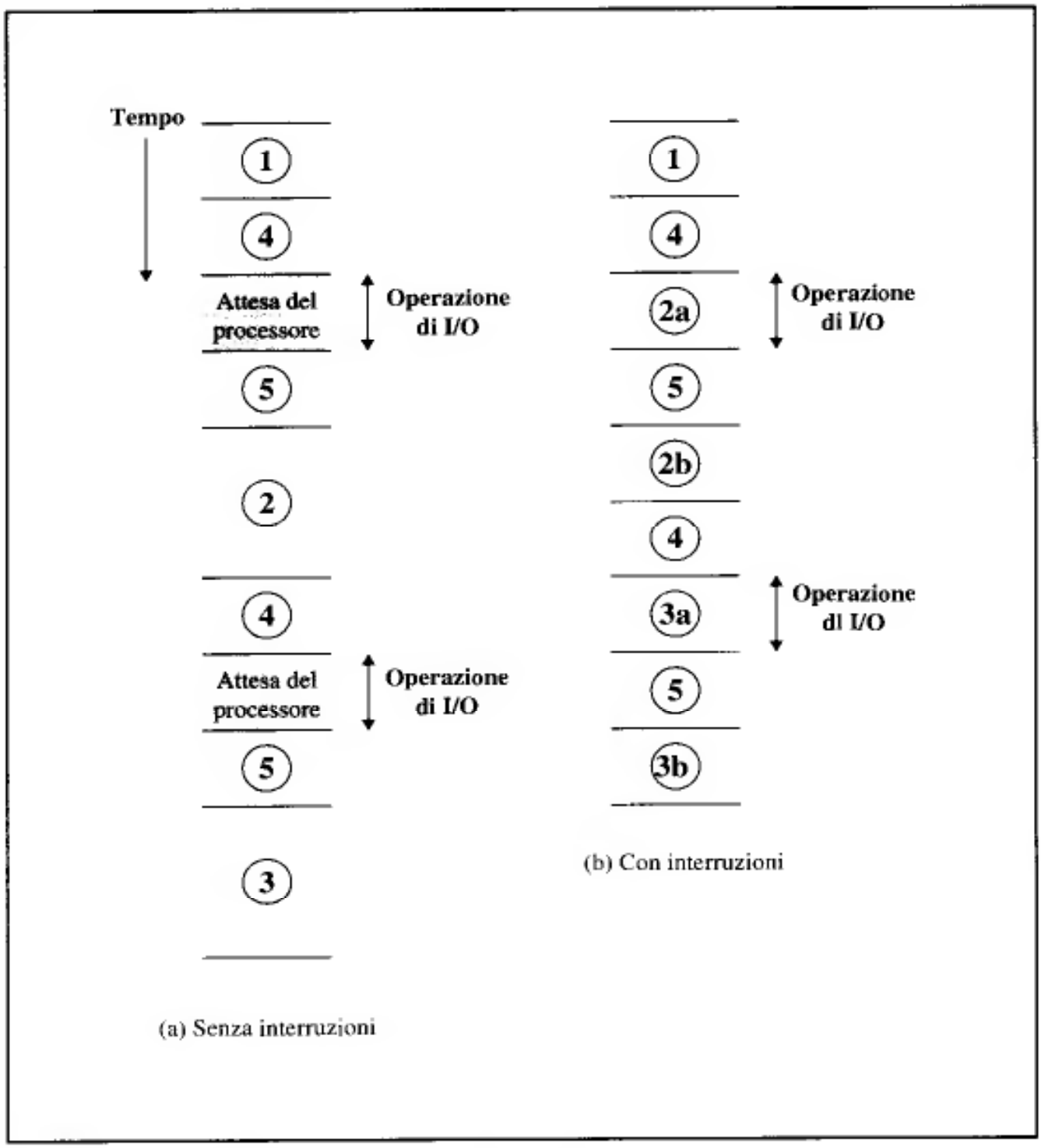


Figura 1.8 *Temporizzazione del programma: attesa di I/O breve*

4. Il processore si prepara a trasferire il controllo alla routine di interrupt. Per iniziare, deve salvare le informazioni necessarie a riprendere il programma corrente dal punto di interruzione. L'informazione minima richiesta è costituita dal PSW e dalla locazione dell'istruzione successiva, contenuta nel program counter, che verranno salvati nello stack di sistema (vedi l'Appendice 1B).
5. Il processore carica il program counter con la locazione di entrata della routine per la gestione dell'interrupt, che risponderà a questa interruzione. A seconda dell'architettura del com-

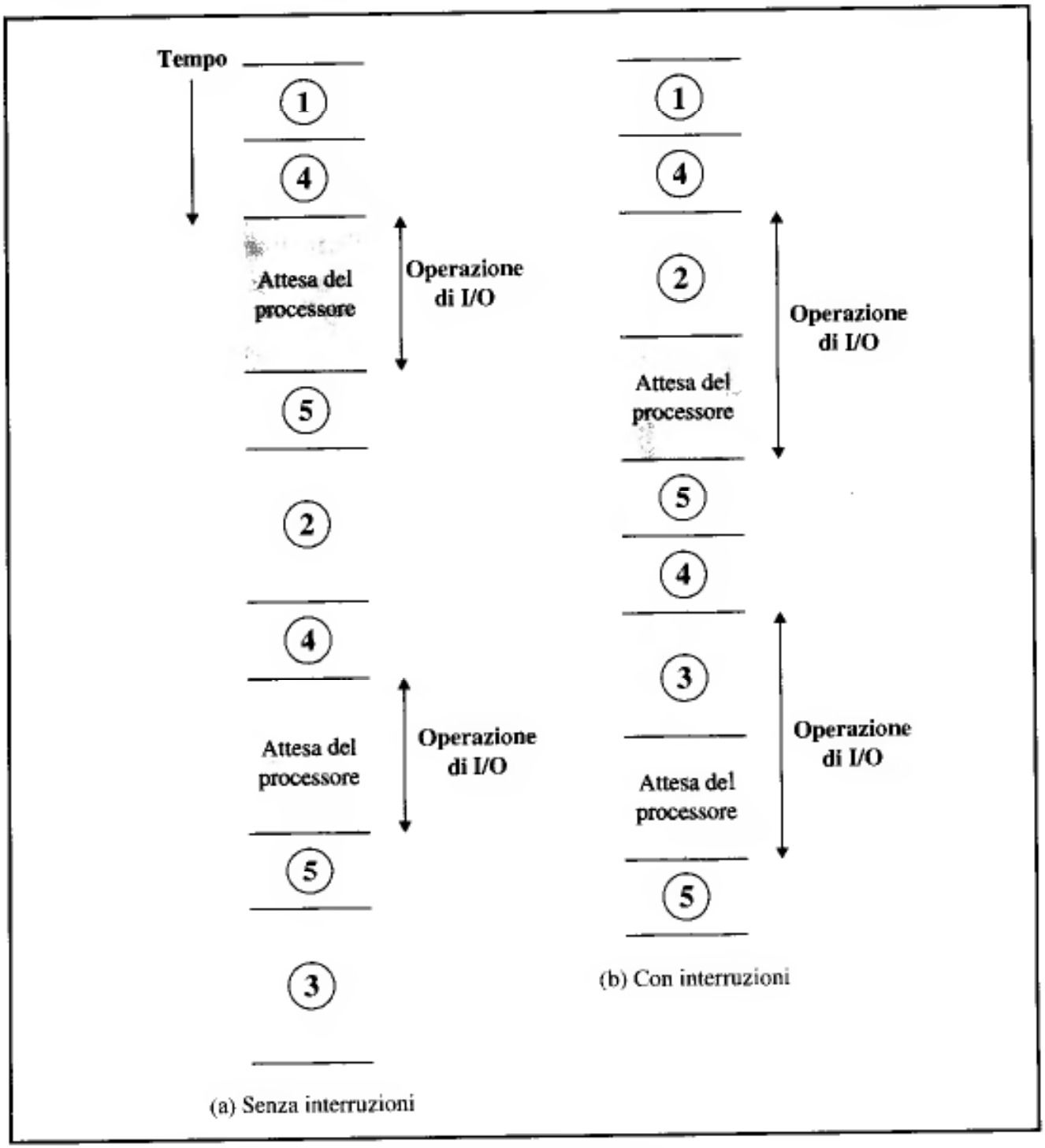


Figura 1.9 *Temporizzazione del programma: attesa di I/O lunga*

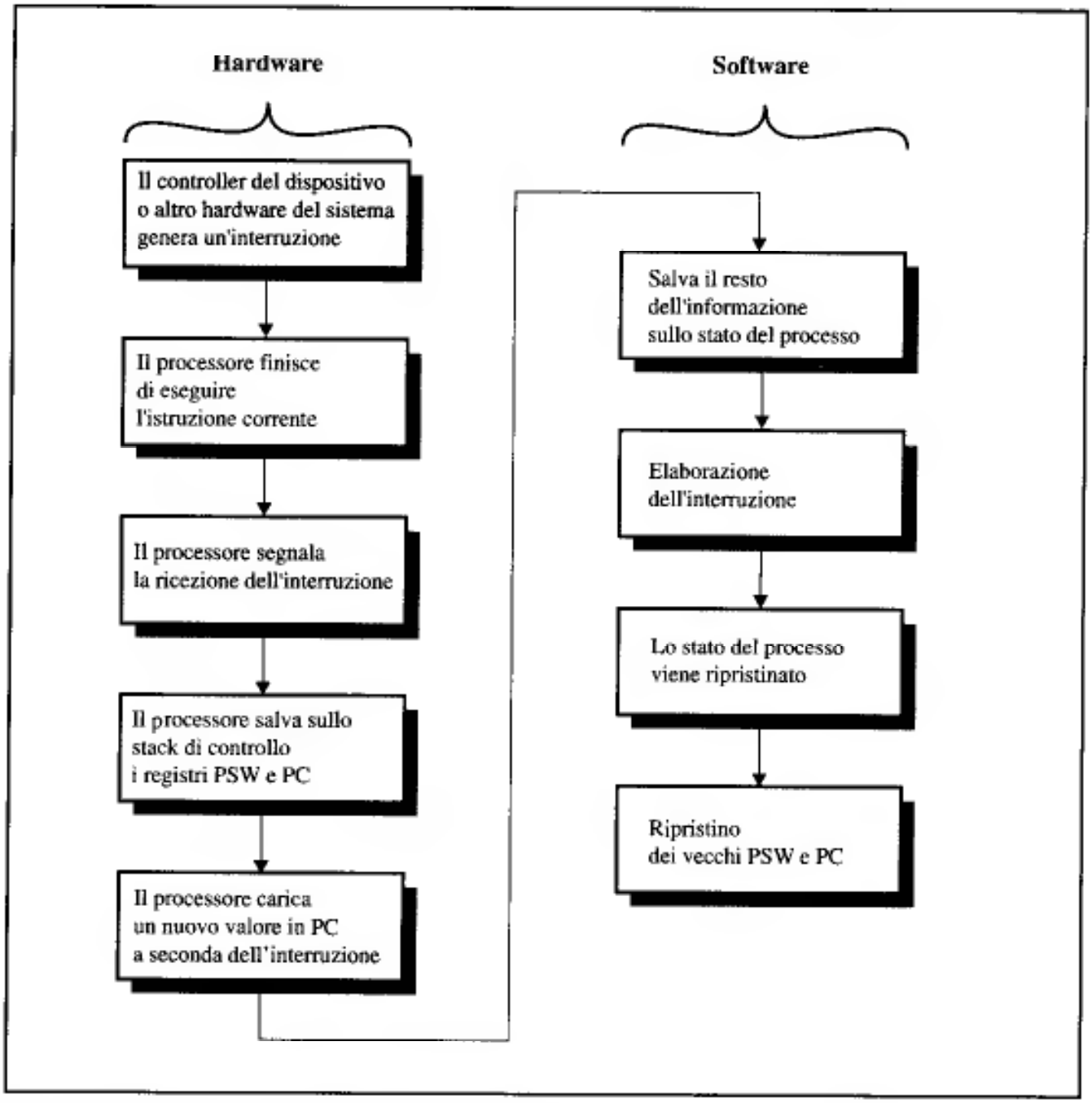


Figura 1.10 *Semplice elaborazione dell'interruzione*

puter e del sistema operativo, è possibile avere un singolo programma, uno per ciascun tipo di interruzione, oppure uno per ciascun dispositivo e ciascun tipo di interruzione. Se esiste più di una routine per la gestione dell'interruzione, il processore deve determinare quale chiamare. Questa informazione può essere inclusa nel segnale di interrupt originale; diversamente, il processore invia una richiesta al dispositivo che ha generato l'interruzione per ottenere una risposta contenente l'informazione necessaria.

Una volta che il program counter è stato caricato, il processore continua con il successivo ciclo d'istruzione, che inizia con un fetch. Poiché il fetch dell'istruzione è determinato dal contenuto del program counter, il controllo viene trasferito al programma di gestione dell'interrupt.

L'esecuzione di questo programma si compone delle seguenti operazioni:

6. A questo punto, il program counter e il PSW relativi al programma interrotto sono stati salvati sullo stack di sistema. Esistono però altre informazioni considerate parte dello stato del programma in esecuzione: in particolare, occorre salvare il contenuto dei registri del processore, che possono essere usati dal gestore dell'interruzione, ed altre eventuali informazioni di stato (di cui si tratterà nel Capitolo 3). Tipicamente, il gestore dell'interruzione inizierà salvando i contenuti di tutti i registri sullo stack. La Figura 1.11a presenta un semplice esempio. Un programma utente viene interrotto dopo l'istruzione alla locazione N; il contenuto di tutti i registri, più l'indirizzo dell'istruzione successiva ($N+1$), sono salvati sullo stack. Lo stack pointer viene aggiornato a puntare alla nuova cima dello stack e il program counter viene aggiornato a puntare all'inizio della routine di servizio dell'interruzione.
7. Il gestore dell'interruzione può ora procedere ad elaborare l'interrupt, esaminando l'informazione di stato relativa all'operazione di I/O, o altri eventi responsabili dell'interruzione. Può inoltre mandare comandi aggiuntivi o acknowledgment al dispositivo di I/O.
8. Completata l'elaborazione dell'interrupt, i valori dei registri salvati sono recuperati dallo stack e rimemorizzati nei registri (ad esempio, vedi la Figura 1.11b).
9. Il passo finale consiste nel recuperare i valori del PSW e del program counter dallo stack. Come risultato, verrà eseguita l'istruzione successiva del programma precedentemente interrotto.

È importante salvare tutte le informazioni di stato relative al programma interrotto per poterle poi ripristinare, in quanto l'interrupt non è una routine chiamata dal programma. Al contrario, l'interruzione può avvenire in ogni momento e in qualsiasi punto del programma utente in esecuzione, pertanto essa è imprevedibile.

Interruzioni multiple

La rassegna appena conclusa ha riguardato unicamente le interruzioni semplici; tuttavia, è possibile che si verifichino interruzioni multiple. Ad esempio, un programma può ricevere dati da una linea di comunicazione e stampare risultati: la stampante genera un'interruzione ogni volta che completa un'operazione di stampa, mentre il controller della linea di comunicazione genera un'interruzione all'arrivo di ogni unità di dati, che potrebbe essere costituita di un singolo carattere o di un blocco, a seconda della natura della comunicazione (*line discipline*). In ogni caso, è possibile che un'interruzione di comunicazione avvenga mentre se ne sta elaborando una della stampante.

Esistono due metodi per trattare le interruzioni multiple. Il primo consiste nel *disabilitare le interruzioni* mentre vengono elaborate: il processore ignorerà il segnale di richiesta di quell'interruzione. Nel caso in cui, durante l'elaborazione di un interrupt, se ne verifichi un secondo, generalmente questo rimane sospeso, e il processore lo controllerà dopo aver abilitato le interruzioni. Pertanto, quando un programma utente è in esecuzione e si verifica un interrupt, le interruzioni vengono disabilitate immediatamente. Completata la routine di gestione dell'interrupt,

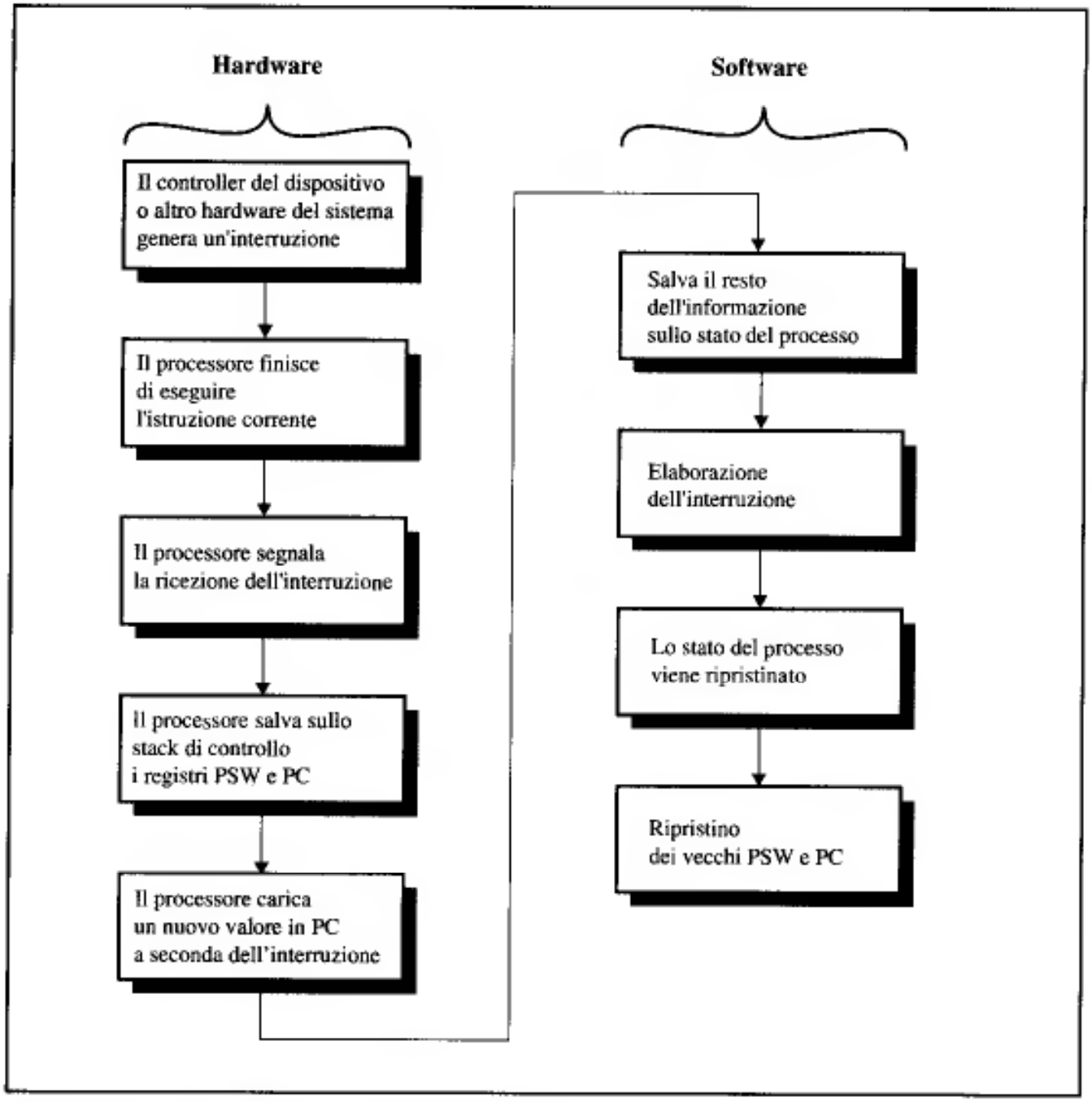


Figura 1.10 *Semplice elaborazione dell'interruzione*

puter e del sistema operativo, è possibile avere un singolo programma, uno per ciascun tipo di interruzione, oppure uno per ciascun dispositivo e ciascun tipo di interruzione. Se esiste più di una routine per la gestione dell'interruzione, il processore deve determinare quale chiamare. Questa informazione può essere inclusa nel segnale di interrupt originale; diversamente, il processore invia una richiesta al dispositivo che ha generato l'interruzione per ottenere una risposta contenente l'informazione necessaria.

Una volta che il program counter è stato caricato, il processore continua con il successivo ciclo d'istruzione, che inizia con un fetch. Poiché il fetch dell'istruzione è determinato dal contenuto del program counter, il controllo viene trasferito al programma di gestione dell'interrupt.

L'esecuzione di questo programma si compone delle seguenti operazioni:

6. A questo punto, il program counter e il PSW relativi al programma interrotto sono stati salvati sullo stack di sistema. Esistono però altre informazioni considerate parte dello stato del programma in esecuzione: in particolare, occorre salvare il contenuto dei registri del processore, che possono essere usati dal gestore dell'interruzione, ed altre eventuali informazioni di stato (di cui si tratterà nel Capitolo 3). Tipicamente, il gestore dell'interruzione inizierà salvando i contenuti di tutti i registri sullo stack. La Figura 1.11a presenta un semplice esempio. Un programma utente viene interrotto dopo l'istruzione alla locazione *N*; il contenuto di tutti i registri, più l'indirizzo dell'istruzione successiva (*N+1*), sono salvati sullo stack. Lo stack pointer viene aggiornato a puntare alla nuova cima dello stack e il program counter viene aggiornato a puntare all'inizio della routine di servizio dell'interruzione.
7. Il gestore dell'interruzione può ora procedere ad elaborare l'interrupt, esaminando l'informazione di stato relativa all'operazione di I/O, o altri eventi responsabili dell'interruzione. Può inoltre mandare comandi aggiuntivi o acknowledgment al dispositivo di I/O.
8. Completata l'elaborazione dell'interrupt, i valori dei registri salvati sono recuperati dallo stack e rimemorizzati nei registri (ad esempio, vedi la Figura 1.11b).
9. Il passo finale consiste nel recuperare i valori del PSW e del program counter dallo stack. Come risultato, verrà eseguita l'istruzione successiva del programma precedentemente interrotto.

È importante salvare tutte le informazioni di stato relative al programma interrotto per poterle poi ripristinare, in quanto l'interrupt non è una routine chiamata dal programma. Al contrario, l'interruzione può avvenire in ogni momento e in qualsiasi punto del programma utente in esecuzione, pertanto essa è imprevedibile.

Interruzioni multiple

La rassegna appena conclusa ha riguardato unicamente le interruzioni semplici; tuttavia, è possibile che si verifichino interruzioni multiple. Ad esempio, un programma può ricevere dati da una linea di comunicazione e stampare risultati: la stampante genera un'interruzione ogni volta che completa un'operazione di stampa, mentre il controller della linea di comunicazione genera un'interruzione all'arrivo di ogni unità di dati, che potrebbe essere costituita di un singolo carattere o di un blocco, a seconda della natura della comunicazione (*line discipline*). In ogni caso, è possibile che un'interruzione di comunicazione avvenga mentre se ne sta elaborando una della stampante.

Esistono due metodi per trattare le interruzioni multiple. Il primo consiste nel *disabilitare le interruzioni* mentre vengono elaborate: il processore ignorerà il segnale di richiesta di quell'interruzione. Nel caso in cui, durante l'elaborazione di un interrupt, se ne verifichi un secondo, generalmente questo rimane sospeso, e il processore lo controllerà dopo aver abilitato le interruzioni. Pertanto, quando un programma utente è in esecuzione e si verifica un interrupt, le interruzioni vengono disabilitate immediatamente. Completata la routine di gestione dell'interrupt,

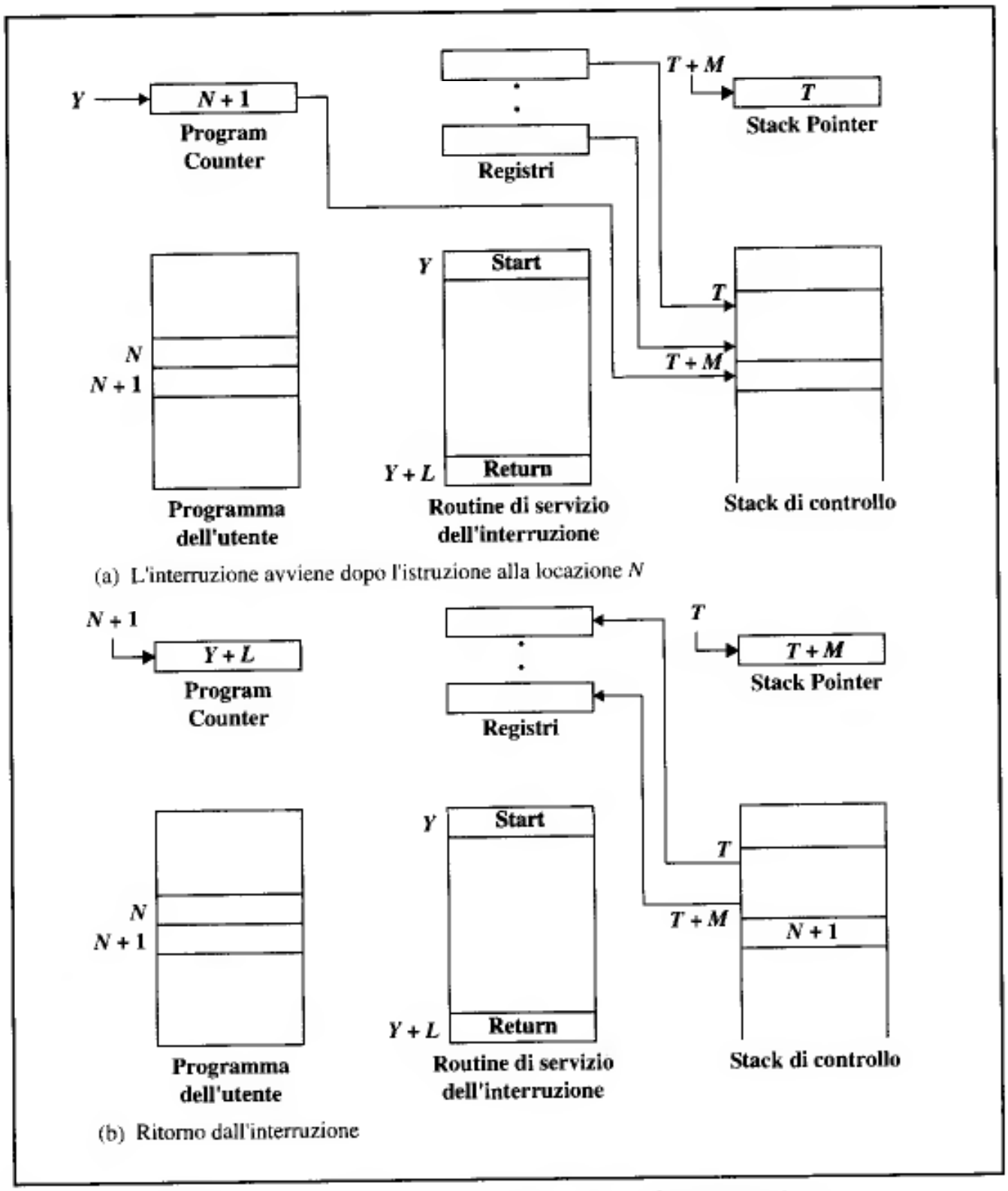


Figura 1.11 *Cambiamenti nella memoria e nei registri dovuti ad un'interruzione*

le interruzioni vengono abilitate prima che si ripristini il programma utente e il processore controlla se siano avvenuti ulteriori interrupt.

Si tratta di un approccio semplice ed efficace, poiché gli interrupt sono gestiti in uno stretto ordine sequenziale (Figura 1.12a), che presenta tuttavia degli svantaggi, in quanto non tiene

conto della priorità relativa e nemmeno della criticità del fattore tempo. Ad esempio, occorre gestire rapidamente gli input in arrivo dalla linea di comunicazione, per far spazio ad altri nuovi: nel caso in cui il primo gruppo di input non sia stato ancora elaborato, è possibile perdere dei dati.

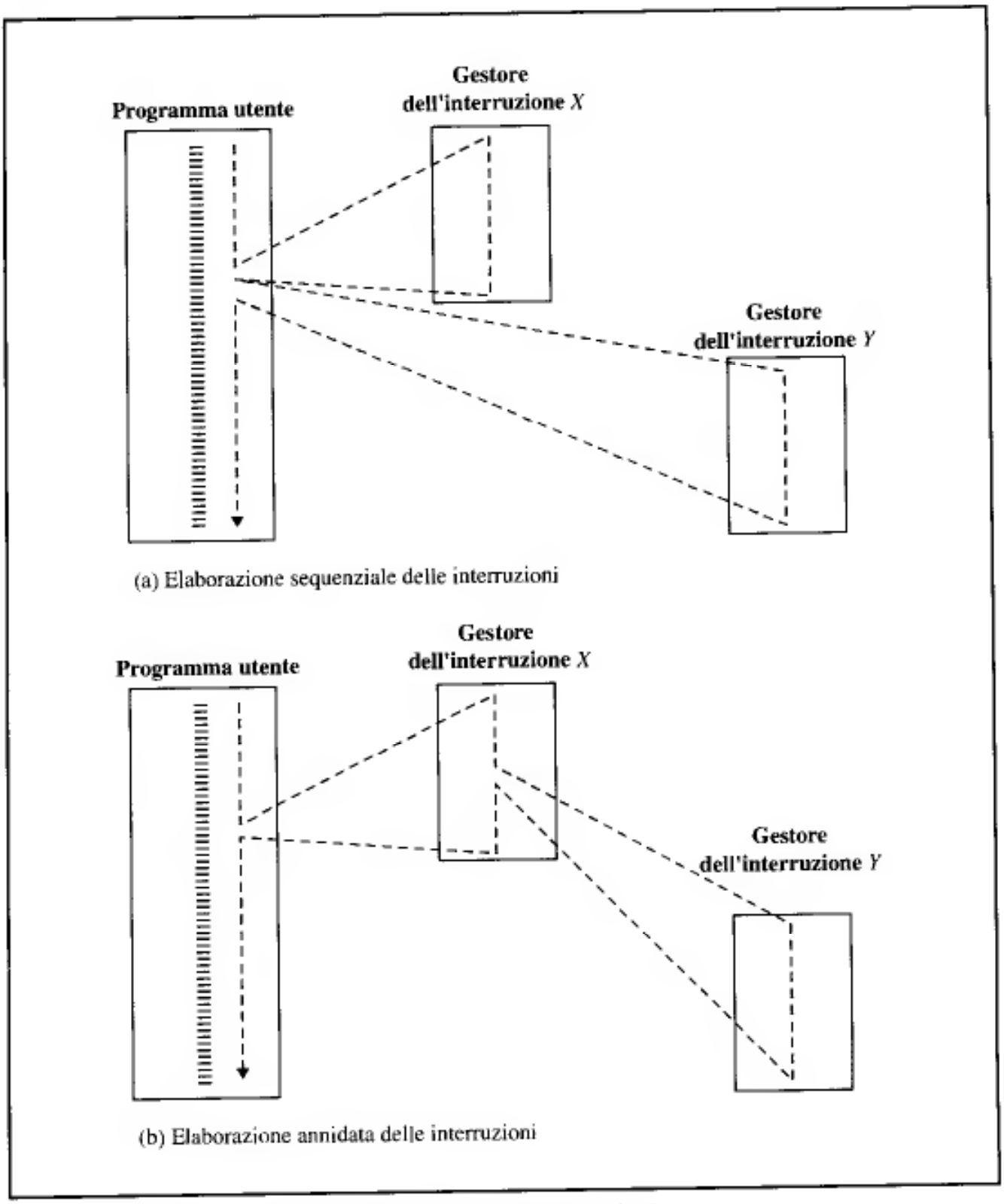


Figura 1.12 *Trasferimento del controllo con interruzioni multiple*

Il secondo approccio consiste nel definire delle priorità, in modo che un'interruzione di più alta priorità interrompa a sua volta un gestore di interruzione a più bassa priorità (Figura 1.12b). Consideriamo come esempio un sistema con tre dispositivi di I/O: una stampante, un disco ed una linea di comunicazione, con priorità crescenti 2, 4 e 5 rispettivamente. La Figura 1.13 illustra una possibile sequenza. Un programma utente inizia al tempo $t = 0$; al tempo $t = 10$ si verifica un interrupt della stampante: l'informazione del programma utente viene posta sullo stack di sistema e l'esecuzione prosegue alla routine di servizio dell'interruzione (ISR, Interrupt Service Routine). Durante tale esecuzione, al tempo $t = 15$ si verifica un interrupt di comunicazione, che viene preso in considerazione poiché la linea di comunicazione ha una priorità più elevata della stampante. L'ISR della stampante viene interrotto, il suo stato viene salvato sullo stack e l'esecuzione prosegue dall'ISR di comunicazione. Mentre questa routine è in esecuzione, si verifica un interrupt del disco ($t = 20$) che, essendo di più bassa priorità, viene semplicemente registrato, mentre l'ISR di comunicazione continua fino al suo naturale completamento.

Quando l'ISR di comunicazione è terminato, ($t = 25$), si ripristina lo stato precedente del processore, in altre parole l'esecuzione dell'ISR della stampante. Comunque, prima che anche una singola istruzione della routine possa essere eseguita, il processore considera l'interrupt del disco, che ha maggiore priorità, ed il controllo passa quindi all'ISR del disco. Solo quando quest'ultima routine viene completata ($t = 35$) si ripristina l'ISR della stampante, al termine del quale (t=40) il controllo ritorna al programma utente.

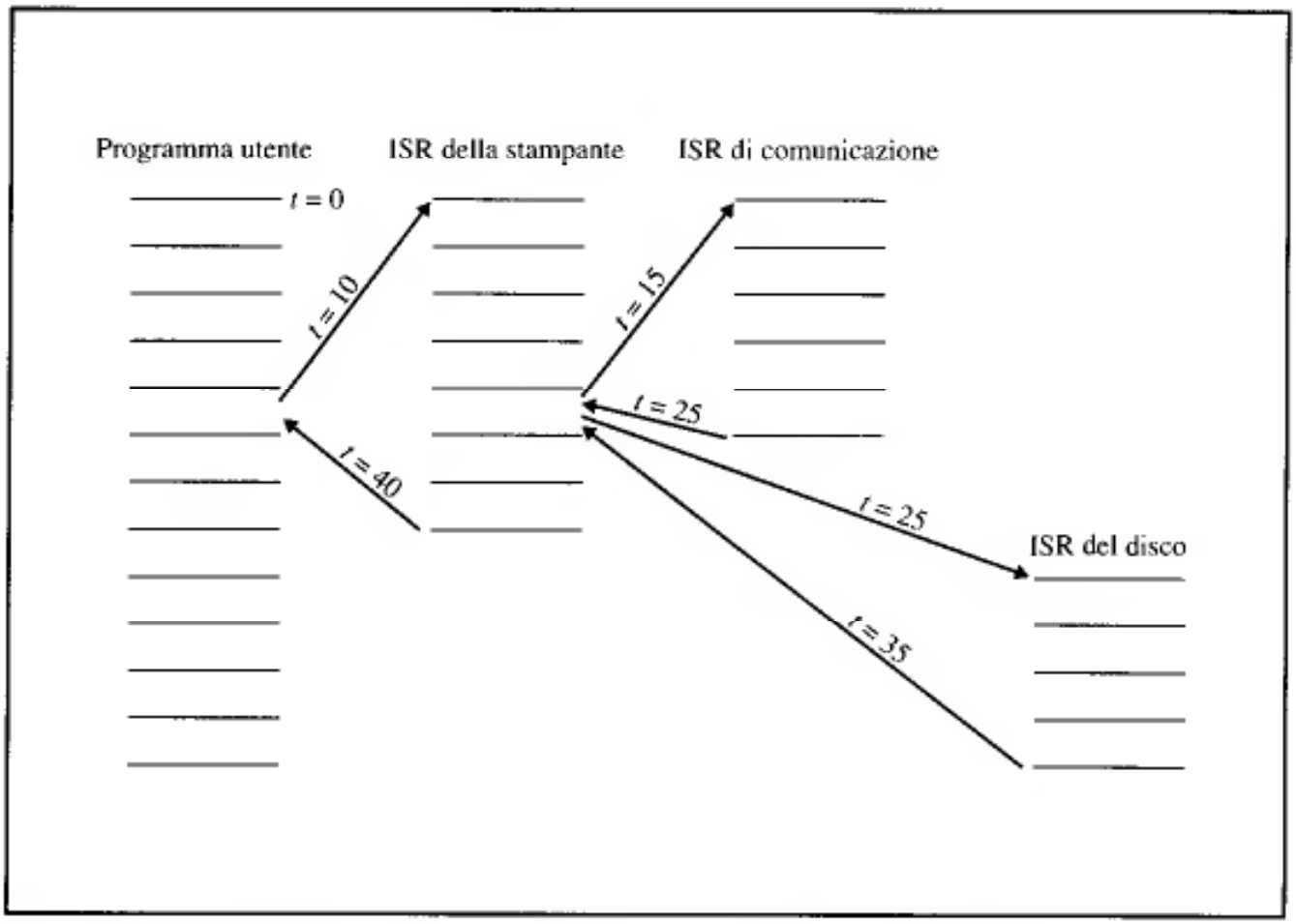


Figura 1.13 *Esempio di sequenza temporale di interruzioni multiple [TANE90]*

Multiprogrammazione

Anche impiegando le interruzioni, il processore non è tuttavia utilizzato al massimo dell'efficienza. A questo proposito, prendiamo di nuovo in esame la Figura 1.9b. Se il tempo richiesto per completare un'operazione di I/O è molto più lungo di quello richiesto dall'esecuzione del codice fra due successive chiamate di I/O (un caso, questo, piuttosto comune), il processore rimarrà inattivo per buona parte del tempo. Una soluzione a questo problema consiste nel permettere a più programmi utente di essere attivi in contemporanea.

Supponiamo, ad esempio, che il processore debba eseguire due programmi: il primo legge semplicemente dei dati dalla memoria, per poi scriverli su un dispositivo esterno; l'altro, invece, è un'applicazione che richiede molto calcolo. Il processore può iniziare il programma di output, eseguire un comando di scrittura su un dispositivo esterno e quindi eseguire l'altra applicazione.

Quando il processore esegue più programmi, la sequenza di esecuzione dipende sia dalla loro priorità relativa, sia dal fatto che siano in attesa per operazioni di I/O. Quando un programma è stato interrotto ed il controllo viene trasferito a un gestore di interrupt, una volta che la relativa routine è terminata, è possibile che il controllo non ritorni immediatamente al programma utente che era in esecuzione in quel momento, ma passi ad altri programmi in attesa con una maggiore priorità: il programma utente interrotto verrà ripristinato per primo nel caso abbia la priorità più elevata. Questo sistema di programmi multipli che si alternano nell'esecuzione è noto come multiprogrammazione e verrà esaminato nel Capitolo 2.

1.5 La gerarchia della memoria

Per quanto riguarda la memoria, è possibile sintetizzare i vincoli inerenti la sua progettazione con queste tre domande: quanto è grande? Quanto è veloce? Quanto è costosa?

Per il primo quesito, non esiste, in un certo senso, una risposta univoca: avendo a disposizione memoria di una certa capacità, verranno probabilmente sviluppate delle applicazioni che la utilizzino al meglio. Il problema della velocità è più semplice da risolvere: per ottenere le migliori prestazioni, la memoria deve essere in grado di supportare la velocità del processore, in modo che, durante l'esecuzione, esso non debba restare in attesa di istruzioni o di operandi. Infine, occorre tenere in considerazione che in un sistema reale, il costo della memoria dev'essere ragionevole, in proporzione a quello delle altre componenti.

Com'è ovvio, c'è un compromesso (*tradeoff*) fra le tre caratteristiche chiave della memoria: costo, capacità e tempo di accesso. Nel corso del tempo, per implementare i sistemi di memoria si è fatto ricorso a tecnologie diverse, che hanno confermato la validità delle seguenti relazioni:

- Tempo di accesso più breve, maggior costo per bit.
- Maggior capacità, minor costo per bit.
- Maggior capacità, maggior tempo di accesso.

Il dilemma che si presenta al progettista è piuttosto evidente: vorrebbe servirsi di tecnologie

che forniscano memorie di grande capacità, perché essa è necessaria, e il costo per bit è basso; al contrario, per soddisfare le prestazioni, il progettista deve utilizzare memorie costose, di capacità relativamente piccola e con tempo di accesso ridotto.

La soluzione di questo dilemma consiste nel non considerare un singolo componente di memoria o una singola tecnologia, ma nell'impiego di una **gerarchia di memoria**, illustrata, in una forma tipica, dalla Figura 1.14.

Scendendo nella gerarchia si verificano i seguenti fenomeni:

a. Diminuisce il costo per bit.

b. Aumenta la capacità.

c. Aumenta il tempo di accesso.

d. Diminuisce la frequenza di accesso alla memoria da parte del processore.

Quindi, memorie più piccole, più costose e più veloci, sono integrate con memorie più grandi, più economiche e più lente. La chiave del successo di tale organizzazione risiede nell'ultimo degli elementi appena elencati, ovvero nella diminuzione della frequenza di accesso. Esamine-

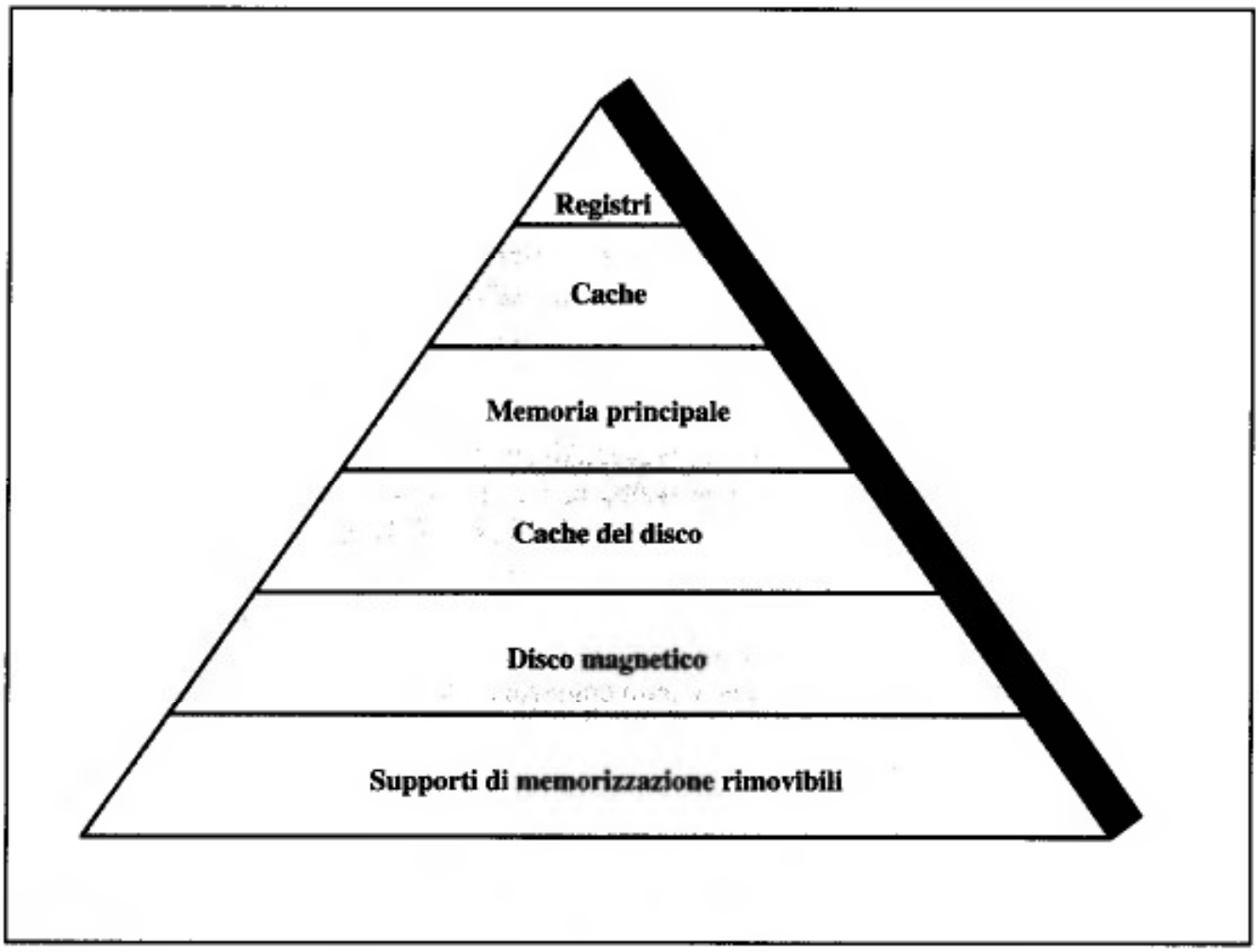


Figura 1.14 *La gerarchia di memoria*

remo più a fondo il concetto in questo capitolo, quando tratteremo della memoria cache e, più avanti, parlando della memoria virtuale; questa che segue, è soltanto una breve spiegazione introduttiva.

Supponiamo che il processore abbia accesso a due livelli di memoria. Il livello 1 contiene 1000 parole e ha un tempo di accesso di 0.1 ms; il livello 2 ne contiene 100000 e ha un tempo di accesso di 1 ms. Se la parola cui si deve accedere è contenuta nel livello 1, il processore vi accede direttamente, ma se la parola si trova nel livello 2, viene prima trasferita al livello 1, e poi acquisita dal processore. Per semplificare, ignoriamo il tempo necessario per determinare in quale livello sia contenuta la parola. La Figura 1.15 illustra la forma generale della curva che rappresenta questa situazione, mostrando il tempo di accesso medio a una memoria a due livelli come una funzione della hit ratio H, dove H è definita come la percentuale degli accessi con successo alla memoria più veloce (cioè la cache), rispetto al numero totale degli accessi. Come si può vedere, per un'alta percentuale di accessi con successo al livello 1, il tempo totale di accesso medio è molto più vicino a quello del livello 1 che a quello del livello 2.

Ricorrere alla gerarchia della memoria è una strategia valida, a patto che si verifichino le quattro condizioni elencate sopra. Grazie all'apporto di tecnologie diverse, esiste ormai un insieme di sistemi di memoria che soddisfa le prime tre condizioni, nonché, di solito, anche la quarta, sulla base del principio noto come *località dei riferimenti* (*locality of reference*) [DENN 68]. Durante l'esecuzione di un programma, i riferimenti alla memoria da parte del processore, sia per le istruzioni come per i dati, tendono a localizzarsi in regioni limitate. Tipicamente, i programmi contengono diversi cicli iterativi e subroutine. Una volta che si è entrati in un ciclo o

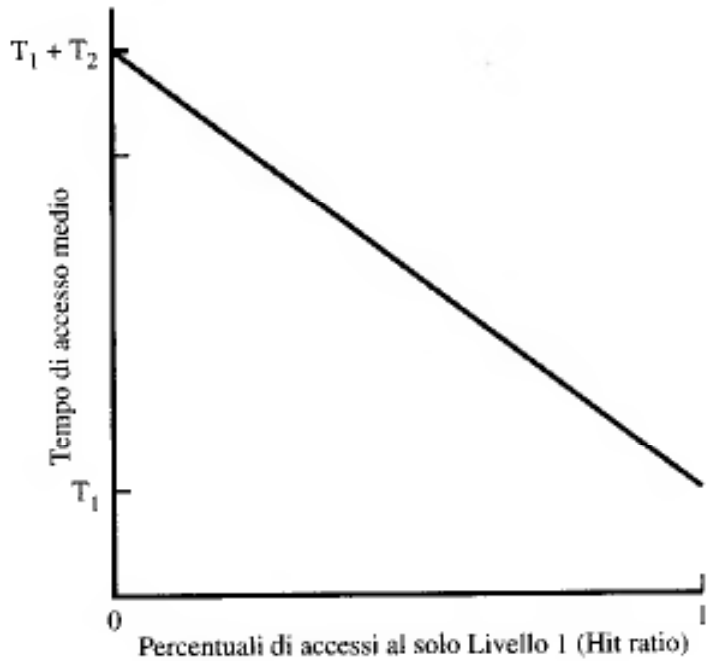


Figura 1.15 *Prestazioni di una semplice memoria a due livelli*

in una subroutine, ci sono riferimenti ripetuti ad un piccolo insieme di istruzioni. Allo stesso modo, operazioni su tabelle o array comportano l'accesso a un insieme localizzato di parole di dati. Su un periodo di tempo lungo, i cluster (*regioni localizzate*) in uso cambiano, mentre su un periodo breve il processore lavora in primo luogo con riferimenti di memoria a cluster fissi.

Di conseguenza, è possibile organizzare i dati lungo la gerarchia in modo tale che la percentuale di accesso a ciascun livello più basso sia sostanzialmente minore di quella del livello superiore. Riconsideriamo ora l'esempio menzionato sopra. Supponiamo che il livello 2 della memoria contenga tutte le istruzioni e i dati del programma: i cluster correnti possono essere temporaneamente posizionati nel livello 1, e, col passare del tempo, uno di questi cluster dovrà essere riportato nel livello 2, per fare spazio a un nuovo cluster in arrivo nel livello 1. In media, comunque, la maggior parte dei riferimenti riguarderà istruzioni e dati contenuti nel livello 1.

Questo procedimento può essere esteso anche a più livelli di memoria. Prendiamo in esame la gerarchia illustrata nella Figura 1.14. La memoria più veloce, più piccola e costosa è quella dei registri interni del processore. Di solito, un processore contiene poche dozzine di tali registri, benché alcune macchine ne contengano centinaia. Due livelli più in basso, troviamo il più importante sistema interno di memoria, ovvero la memoria principale, dove ciascuna locazione ha un unico indirizzo, e la maggior parte delle istruzioni macchina si riferisce ad uno o più indirizzi della memoria principale. Questa, di solito, viene estesa con una memoria cache più piccola e più veloce, che generalmente non è visibile né al programmatore, né al processore. La cache è un dispositivo per effettuare movimenti di dati fra la memoria principale e i registri del processore, al fine di migliorare le prestazioni.

I tre tipi di memoria appena descritti sono di norma volatili e tutti e tre impiegano la tecnologia dei semiconduttori, in quanto la memoria a semiconduttori è disponibile in una varietà di tipi, che differiscono per velocità e costo. I dati vengono memorizzati permanentemente su dispositivi esterni di memoria di massa, di cui i più comuni sono gli hard disk e i supporti rimovibili, ad esempio dischi estraibili, nastri e memorie ottiche. La memoria esterna non volatile, definita anche secondaria o ausiliaria, è usata per memorizzare file di programmi e di dati. Di solito, è visibile al programmatore solo in termini di file e record, e non di singoli byte o parole. I dischi forniscono inoltre un'estensione alla memoria principale, nota come memoria virtuale, che esamineremo nel Capitolo 8.

Alla gerarchia di memoria è possibile aggiungere, via software, dei livelli addizionali, ad esempio usando una porzione di memoria principale come buffer per memorizzare dati temporanei letti dal disco. Tale tecnica, definita talvolta disk cache (di cui ci occuperemo in dettaglio nel Capitolo 11), migliora le prestazioni in due modi:

- Si raggruppano le scritture su disco: invece di numerosi piccoli trasferimenti di dati, ne abbiamo pochi, ma di maggiori dimensioni. Ne derivano migliori prestazioni del disco e un minor carico sul processore.
- Alcuni dati destinati ad essere scritti, possono essere referenziati da un programma prima della successiva scrittura definitiva su disco. In questo caso, i dati vengono rapidamente recuperati dalla cache software, invece che, con maggior lentezza, dal disco.

L'Appendice 1A esamina le implicazioni relative alle prestazioni delle memorie a più livelli.

1.6 Memoria cache

Pur essendo invisibile al sistema operativo, la memoria cache interagisce con altri dispositivi hardware per la gestione della memoria; inoltre, molti dei principi utilizzati negli schemi di memoria virtuale sono applicati anche alla memoria cache.

Motivazione

In tutti i cicli di istruzione, il processore accede alla memoria almeno una volta per prelevare l'istruzione e spesso ripete il procedimento, per prelevare gli operandi e/o memorizzare i risultati. La velocità con cui il processore esegue le istruzioni è ovviamente limitata dal tempo di accesso alla memoria, il che ha sempre costituito un notevole problema, a causa della perdurante sfasatura tra le velocità del processore e della memoria principale. Nel corso degli anni, infatti, la velocità del processore è cresciuta molto più rapidamente di quella di accesso alla memoria, sicché dobbiamo confrontarci col tradeoff tra velocità, costo e dimensioni. In teoria, si dovrebbe costruire la memoria principale con la stessa tecnologia dei registri del processore, e con tempi di ciclo di memoria comparabili a quelli del processore stesso. Tale strategia, però, si è sempre

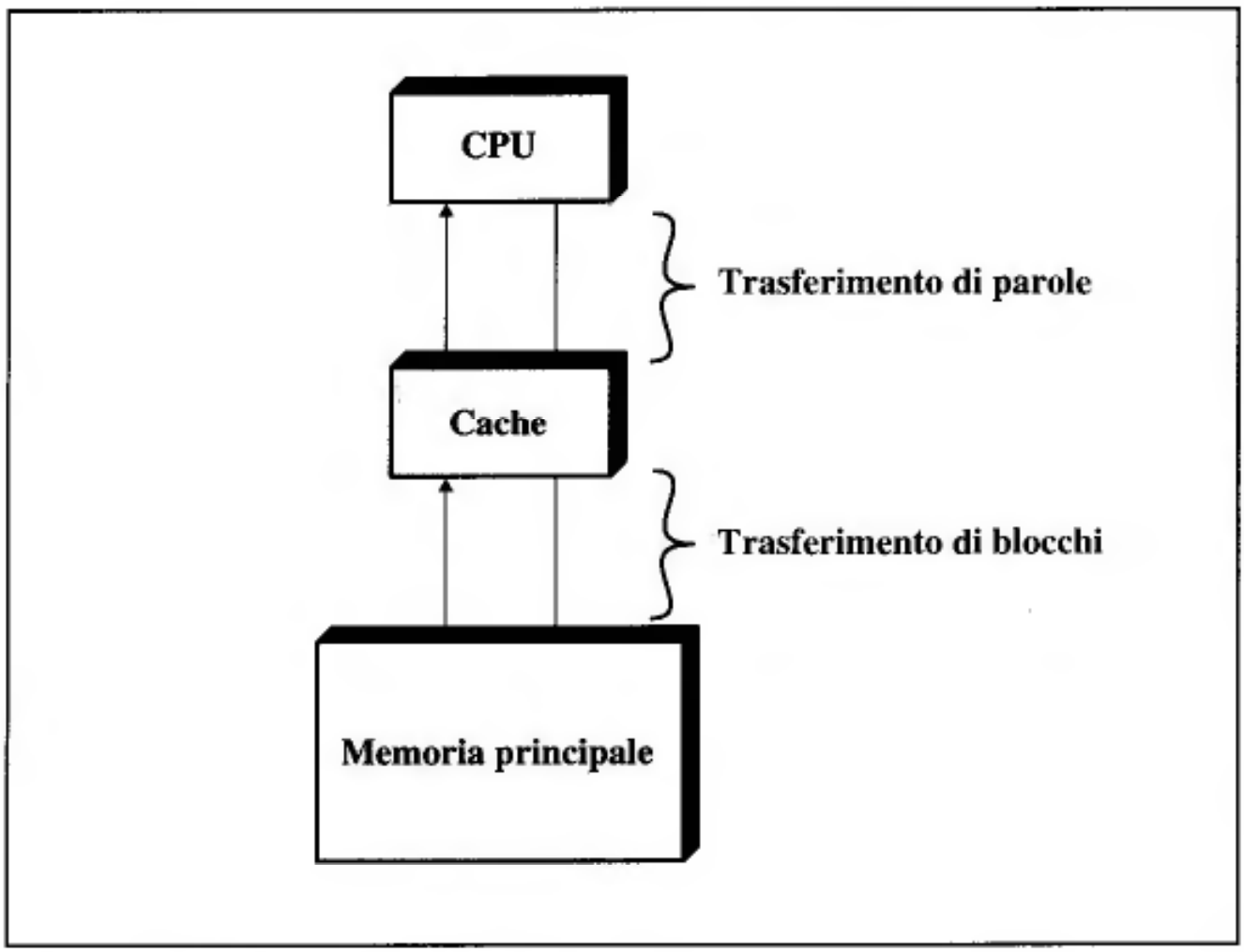


Figura 1.16 *Memoria cache e memoria principale*

rivelata troppo costosa. La soluzione sta nello sfruttare il principio della località, inserendo una memoria piccola e veloce, cioè la cache, fra il processore e la memoria principale.

Principi della cache

Obiettivo della cache è fornire una memoria la cui velocità approssimi quella delle memorie più veloci, e che sia allo stesso tempo abbastanza grande, però al costo delle memorie a semiconduttori più economiche. Il concetto è illustrato nella Figura 1.16, dove una memoria principale, relativamente grande e più lenta, è presente insieme a una memoria cache più piccola e veloce. La cache contiene una copia di una porzione della memoria principale: quando il processore tenta di leggere una parola di memoria, un test determina se essa si trovi nella cache; in caso affermativo, la parola è inviata al processore. In caso contrario, un blocco di memoria principale, comprendente un numero fisso di parole, viene letto nella cache, e la parola è inviata al processore. A causa del fenomeno della località dei riferimenti, quando un blocco di dati viene caricato nella cache per soddisfare un singolo riferimento alla memoria, è probabile che i futuri riferimenti siano relativi ad altre parole nel blocco.

La Figura 1.17 illustra la struttura di un sistema cache/memoria principale. Quest'ultima

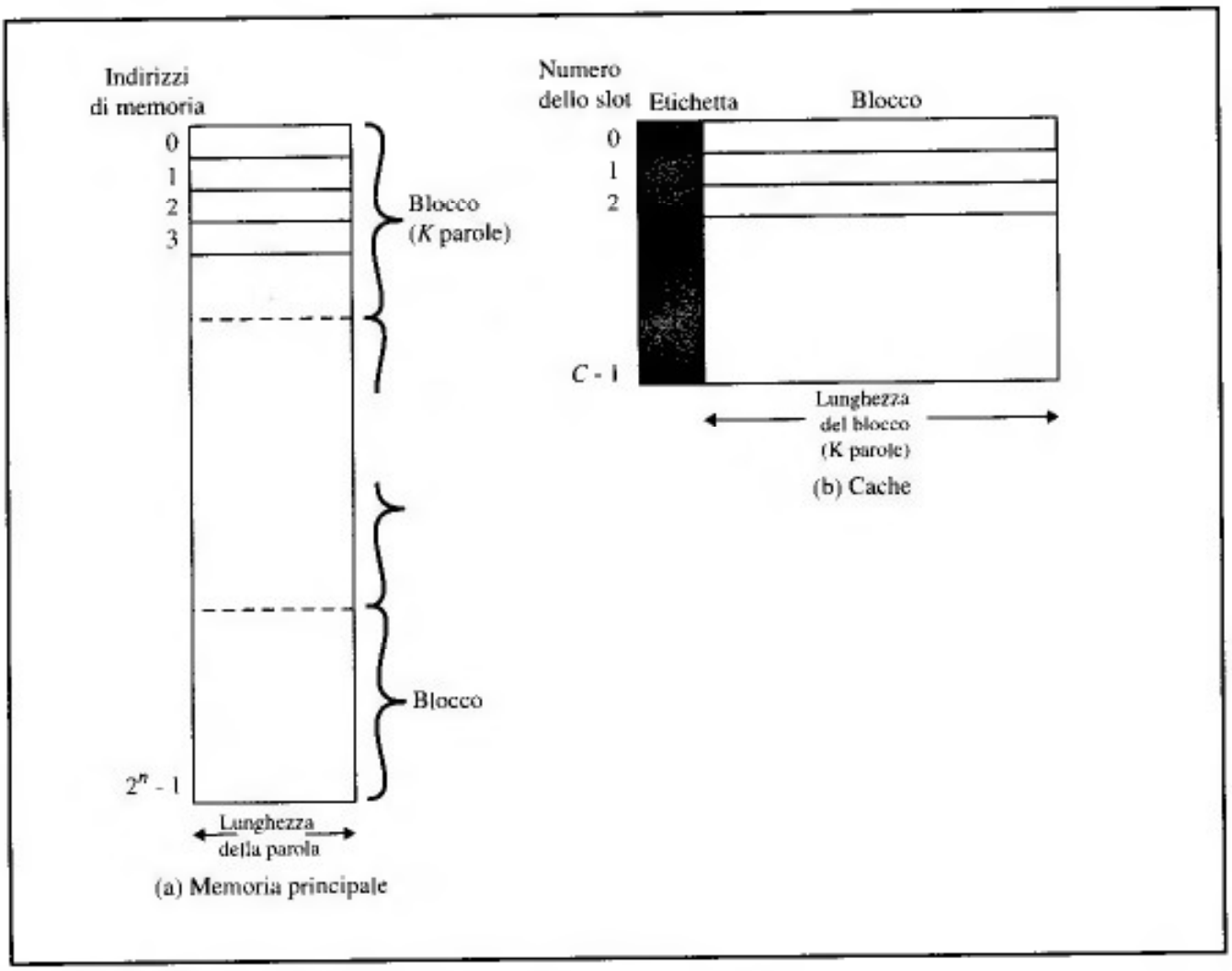


Figura 1.17 *Struttura della cache e della memoria principale*

consiste di 2^n parole indirizzabili, ognuna delle quali possiede un unico indirizzo a n-bit. Ai fini del mapping, si può dividere la memoria in blocchi di lunghezza fissata di K parole ciascuno, per un totale di $M = 2^n/K$ blocchi. La cache consiste di C slot di K parole ciascuno, e il numero dei suoi slot è considerevolmente minore di quello della memoria principale ($C << M$). Un sottoinsieme dei blocchi della memoria principale si trova negli slot della cache. Se viene letta una parola in un blocco di memoria che non si trova nella cache, quel blocco viene trasferito in uno dei suoi slot. Poiché esistono più blocchi che slot, un singolo slot non può essere sempre e soltanto dedicato ad un blocco particolare; perciò ciascuno include un'etichetta, che identifica quale particolare blocco è memorizzato attualmente: l'etichetta di solito e' costituita dai bit di ordine superiore dell'indirizzo.

La Figura 1.18 illustra l'operazione di lettura. Il processore genera l'indirizzo (RA) di una parola che deve essere letta: se è contenuta nella cache, è inviata al processore. Diversamente, il blocco che contiene la parola viene caricato nella cache e la parola inviata al processore.

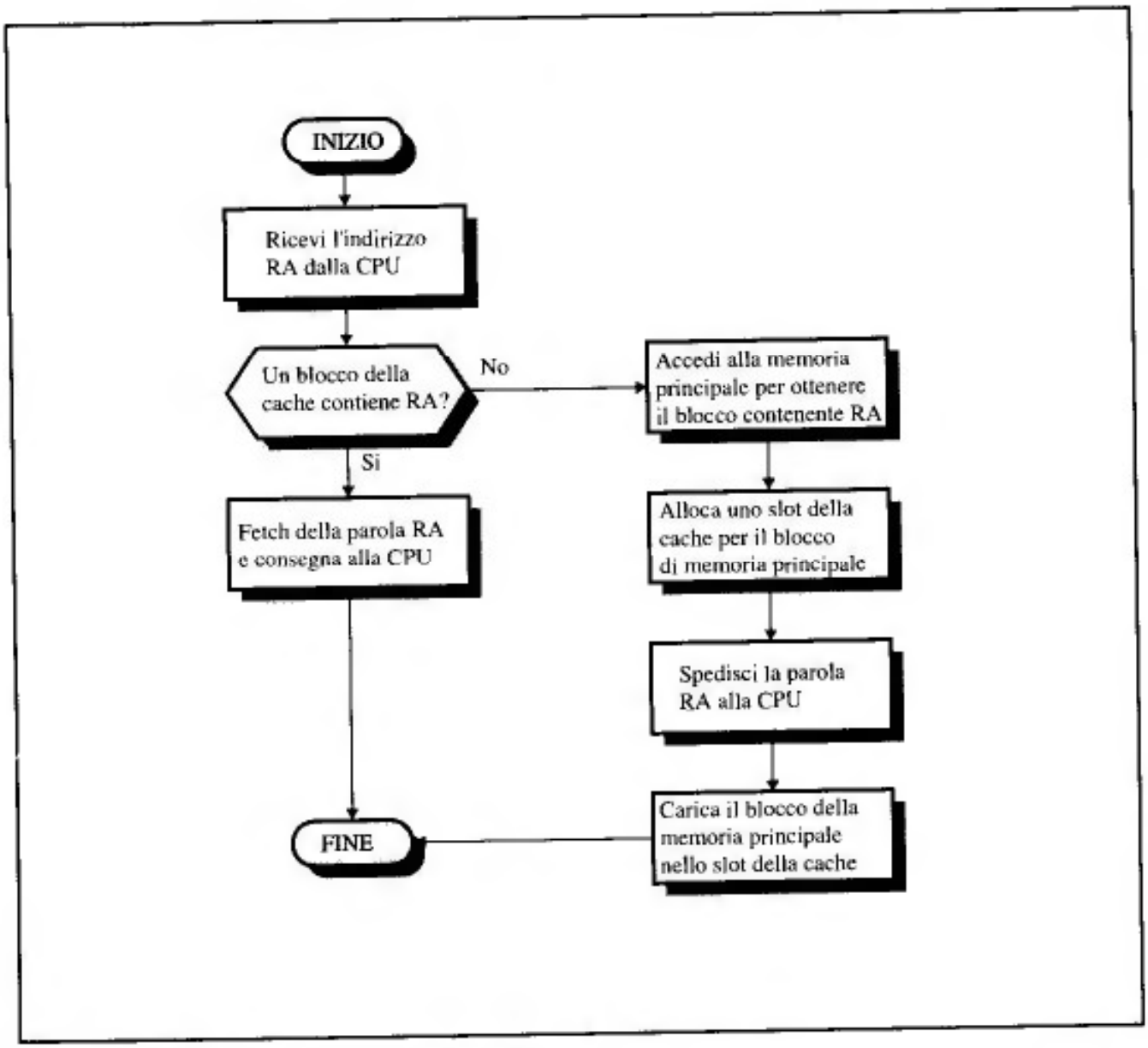


Figura 1.18 *Operazione di lettura della cache*

Progettazione della cache

Spiegare in dettaglio come si progetti una cache non è fra gli obiettivi di questo testo, pertanto qui di seguito saranno brevemente illustrati solo gli elementi chiave. Le problematiche inerenti la progettazione della memoria virtuale e della cache del disco sono simili, e ricadono in queste categorie:

- Dimensione della cache
- Dimensione dei blocchi
- Funzione di mappatura
- Algoritmo di rimpiazzo
- Politica di scrittura.

Abbiamo già trattato il problema della **dimensione della cache**, osservando come cache ragionevolmente piccole influiscano in modo significativo sulle prestazioni. Un altro problema è quello della **dimensione dei blocchi**, cioè delle unità di dati scambiate fra la cache e la memoria principale. Man mano che la dimensione dei blocchi aumenta, la hit ratio dapprima cresce a causa del principio di località, cioè per l'alta probabilità che i dati vicini a parole referenziate vengano verosimilmente referenziati in un immediato futuro: aumentando la dimensione del blocco, nella cache si caricano più dati utili. Se però aumentiamo ancora la dimensione del blocco, la hit ratio inizierà a decrescere, quando la probabilità di usare i dati prelevati di recente diventerà minore di quella di riusare i dati rimossi dalla cache per far posto al nuovo blocco.

Quando un nuovo blocco di dati viene letto nella cache, la **funzione di mappatura** determina quale locazione della cache stessa esso occuperà. Progettando tale funzione, occorre tener conto di due vincoli. In primo luogo, quando un blocco viene letto, un altro può dover essere rimpiazzato, e si dovrà cercare di ridurre al minimo la probabilità di rimpiazzare un blocco necessario in un prossimo futuro. Quanto più è flessibile la funzione di mappatura, tanto più avremo modo di progettare un algoritmo di rimpiazzo che massimizzi la hit ratio. In secondo luogo, quanto più flessibile è la funzione di mappatura, tanto più complessi saranno i circuiti per determinare se un certo blocco si trovi nella cache.

L'**algoritmo di rimpiazzo** sceglie, entro i vincoli della funzione di mappatura, quale blocco rimpiazzare. Sarebbe preferibile individuare il blocco probabilmente meno necessario nel prossimo futuro; non essendo però possibile identificarlo, una strategia abbastanza efficace consiste nel rimpiazzare il blocco che è stato nella cache più a lungo senza alcun riferimento, mediante l'algoritmo LRU (*usato meno di recente*, Least Recently Used). Per identificare tale blocco sono necessari meccanismi hardware.

Qualora i contenuti di un blocco nella cache siano stati modificati, prima di rimpiazzarlo occorre riscriverlo nella memoria principale. La **politica di scrittura** determina quando debba avvenire l'operazione di scrittura in memoria. Ad un estremo, è possibile scrivere ogni volta che il blocco è aggiornato, all'altro estremo, solo quando esso viene rimpiazzato. Questa seconda politica minimizza le operazioni di scrittura in memoria, lasciando però la memoria principale in uno stato non aggiornato. Ciò può interferire in caso di operazioni con processori multipli e di accessi diretti alla memoria da parte di moduli di I/O.

1.7 Tecniche di comunicazione di I/O

Le operazioni di I/O possono essere di tre tipi:

- I/O programmato
- I/O interrupt-driven (*guidato da interruzioni*)
- DMA (*accesso diretto alla memoria*, Direct Memory Access).

I/O programmato

Quando il processore esegue un programma e incontra un'istruzione di I/O, la esegue inviando un comando all'appropriato modulo di I/O. Con l'I/O programmato, il modulo di I/O esegue l'azione richiesta, quindi imposta i bit appropriati nel registro di stato di I/O. Il modulo di I/O non esegue ulteriori azioni per avvertire il processore e, in particolare, non interrompe il processore stesso, al quale pertanto spetta il compito di controllare periodicamente lo stato del modulo, fino a verificare che l'azione sia completata.

Con questa tecnica è il processore ad estrarre i dati dalla memoria principale per le operazioni di output e a memorizzarveli per le operazioni di input. Il software di I/O è scritto in modo che il processore esegua istruzioni che gli diano il diretto controllo dell'operazione di I/O stessa, incluso il controllo dello stato del dispositivo, l'invio di comandi di lettura e scrittura e trasferimento dei dati. Quindi, l'insieme delle istruzioni di I/O comprende i seguenti tipi:

- **Istruzioni di controllo**: per attivare un dispositivo esterno e determinare l'operazione da effettuare. Ad esempio, ordinare ad un nastro magnetico di riavvolgersi o di muoversi in avanti di un record.
- **Istruzioni di test**: per effettuare il test di varie condizioni di stato associate ai moduli di I/O e loro periferiche.
- **Operazioni di lettura e scrittura**: per trasferire dati fra i registri del processore e i dispositivi esterni.

La Figura 1.19a fornisce un esempio di I/O programmato per leggere in memoria un blocco di dati provenienti da un dispositivo esterno: ad esempio, un record da un nastro. I dati vengono letti una parola (cioè 16 bit) alla volta; per ciascuna parola letta, il processore deve rimanere in un ciclo di controllo di stato finché determina che la parola è disponibile nel registro dei dati del modulo di I/O. Il diagramma di flusso della Figura 1.19a evidenzia lo svantaggio principale di questa tecnica: si tratta, infatti, di un procedimento oneroso, che tiene il processore inutilmente occupato.

I/O Interrupt - driven

Con l'I/O programmato, il processore deve attendere a lungo prima che il modulo di I/O interessato sia pronto per ricevere o per trasmettere nuovi dati e, durante questa attesa, deve

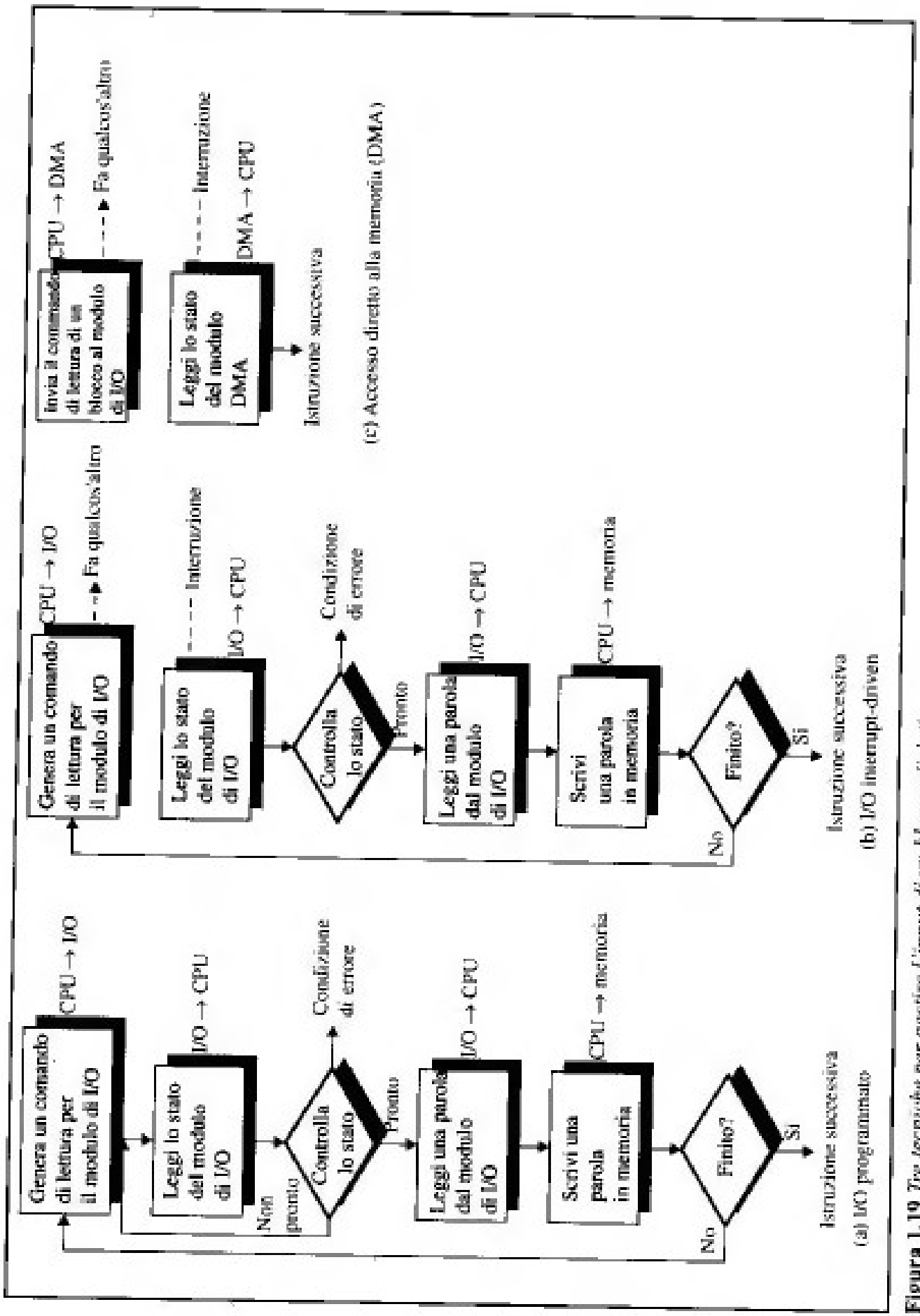


Figura 1.19 *Tre tecniche per gestire l'input di un blocco di dati*

ripetutamente interrogare lo stato di tale modulo di I/O. Ne consegue una forte diminuzione delle prestazioni dell'intero sistema.

In alternativa, il processore può inviare un comando di I/O ad un modulo, proseguendo quindi a svolgere altro lavoro utile senza mettersi in attesa. Il modulo di I/O interromperà il processore per richiedere un servizio quando sarà pronto a scambiare dati con il processore stesso, che eseguirà, come s'è detto prima, il trasferimento dei dati, per poi riprendere la sua elaborazione precedente.

Prendiamo in esame questo processo, partendo dal modulo di I/O. Il modulo di I/O riceve in ingresso un comando READ (*di lettura*) dal processore, quindi inizia a leggere i dati dalla periferica associata. Una volta che i dati si trovano nel registro dei dati del modulo, questo invia un segnale di interruzione al processore su una linea di controllo. Il modulo attende finché i suoi dati sono richiesti dal processore, quindi li invia sul bus dei dati ed è pronto per un'altra operazione di I/O.

Dal punto di vista del processore, l'operazione di input si svolge come segue. Il processore invia un comando READ, quindi salva il contesto (cioè il program counter e i registri del processore) del programma corrente e passa ad eseguire un'altra operazione (è possibile, infatti, che il processore stia lavorando su diversi programmi nel medesimo tempo). Alla fine di ciascun ciclo di istruzione, il processore controlla se esistano interrupt pendenti (Figura 1.7). Quando avviene l'interruzione dal modulo di I/O, il processore salva il contesto del programma che sta eseguendo correntemente, ed inizia ad eseguire una routine per la gestione di tale interruzione. In questo caso, il processore legge la parola di dati dal modulo di I/O e la salva in memoria. Infine, ripristina il contesto del programma che ha chiamato il comando di I/O (oppure di qualche altro programma) e riprende l'esecuzione.

La Figura 1.19b illustra l'uso dell'I/O interrupt-driven per la lettura di un blocco di dati. Confrontiamola con la Figura 1.19a: l'I/O interrupt-driven si rivela più efficiente di quello programmato, perché elimina le attese non necessarie, tuttavia, impegna il processore in modo rilevante, in quanto ogni parola di dati che va dalla memoria al modulo di I/O o viceversa, deve passare attraverso il processore stesso.

Quasi sempre, in un computer, esistono numerosi moduli di I/O, pertanto il processore necessita di meccanismi per determinare quale dispositivo abbia causato l'interruzione e per decidere, nel caso di interrupt multipli, quale gestire per primo. In alcuni sistemi esistono molte linee di interrupt, cosicché ciascun modulo di I/O invia segnali su una linea differente, contrassegnata da una differente priorità. In alternativa, è possibile avere una linea di interrupt singola, utilizzando però linee addizionali per determinare l'indirizzo del dispositivo: anche in questo caso, a dispositivi differenti corrispondono priorità differenti.

Accesso diretto alla memoria

L'I/O interrupt-driven, benché più efficiente del semplice I/O programmato, richiede tuttavia l'intervento attivo del processore per effettuare ogni trasferimento di dati fra la memoria e un modulo di I/O, e ciascun trasferimento di dati deve passare attraverso il processore. Quindi, ambedue le forme di I/O soffrono di due limitazioni:

1. Il tasso di trasferimento di I/O è limitato dalla velocità con cui il processore può testare e servire un dispositivo.

2. Il processore è obbligato a gestire il trasferimento di I/O, eseguendo un certo numero di istruzioni per ciascun trasferimento.

Per spostare una quantità di dati considerevole, bisogna ricorrere ad una tecnica più efficiente, e cioè l'accesso diretto alla memoria. La funzione DMA può essere eseguita da un modulo separato sul bus di sistema o incorporata in un modulo di I/O. In entrambi i casi, la tecnica è la seguente. Quando il processore desidera leggere o scrivere un blocco di dati, esegue un comando per il modulo DMA inviandogli le seguenti informazioni:

- Richiesta di lettura o scrittura.
- Indirizzo del dispositivo di I/O.
- Locazione di partenza della memoria da cui leggere o scrivere.
- Numero di parole da leggere o scrivere.

Il processore a questo punto può continuare ad eseguire altre istruzioni, avendo delegato quest'operazione di I/O al modulo DMA, che trasferirà l'intero blocco di dati, una parola alla volta, direttamente alla memoria o dalla memoria, senza passare attraverso il processore. Completato il trasferimento, il modulo DMA manda un segnale di interruzione al processore, che risulta impegnato solo all'inizio e alla fine del trasferimento (Figura 1.19c).

Il modulo DMA deve acquisire il controllo del bus per trasferire i dati, entrando talvolta in competizione col processore che, se necessita del bus, deve restare in attesa. Non si tratta però di un'interruzione, in quanto il processore non salva un contesto, né esegue altre operazioni, limitandosi a rimanere in pausa per un ciclo di bus. L'effetto complessivo è di rallentare il processore durante un trasferimento DMA, tuttavia, per un trasferimento I/O di molte parole, il DMA è di gran lunga più efficiente dell'I/O interrupt-driven o programmato.

1.8 Letture raccomandate

[STAL96] tratta in modo approfondito gli argomenti contenuti questo capitolo. Esistono comunque molti altri testi sull'organizzazione e l'architettura dei computer, tra i quali i più validi sono i seguenti: [PATT94] è una rassegna completa; [HENN90], degli stessi autori, è un testo più avanzato, che sottolinea gli aspetti quantitativi dell'architettura. [HERZ96] tratta in dettaglio l'implementazione della logica digitale dei processori e mostra come siano costruiti i componenti chiave dei processori all'interno dell'architettura complessiva. [KAIN96] fornisce una panoramica più avanzata sull'architettura dei computer. Molti testi sull'architettura degli elaboratori e sui sistemi operativi forniscono una trattazione dei principi base degli interrupt; una descrizione particolarmente chiara ed esauriente si trova in [BECK90].

BECK90 Beck, L. *System Software*. Reading, MA: Addison Wesley, 1990

HENN90 Hennessy, J., e Patterson, D. *Computer Architecture: A Quantitative Approach*.

San Mateo, CA: Morgan Kauffman, 1990. Edizione italiana *Architetture dei calcolatori* Zanichelli,1993.

HERZ96 Herzog, J. *Design and Organization of Computing Structures*. Wilsonville, OR: Franklin, Beedle, and Associates, 1996.

KAIN96 Kain, R. *Advanced Computer Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1996.

PATT94 Patterson, D., e Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufman, 1994. Edizione italiana *Struttura, organizzazione e progetto dei calcolatori*, Jackson Libri, 1999.

STAL96 Stallings, W. *Computer Organization and Architecture*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1996.

1.9 Problemi

1.1 La macchina ipotetica della Figura 1.3 ha anche due istruzioni I/O:

0011= Carica AC da I/O

0111= Memorizza AC in I/O

In questi casi l'indirizzo a 12 bit identifica un particolare dispositivo esterno. Rappresentare l'esecuzione del seguente programma, utilizzando il formato della Figura 1.4:

a. Carica AC dal dispositivo 5 (si assuma che il valore successivo sia 3).

b. Aggiungi il contenuto della locazione di memoria 940 (si assuma che il valore corrente sia 2)

c. Memorizza AC nel dispositivo 6.

Si assuma che il valore successivo recuperato dal dispositivo 5 sia 3 e che la locazione 940 contenga il valore 2.

1.2 Si consideri un ipotetico microprocessore a 32 bit con istruzioni a 32 bit composte di due campi: il primo byte contiene l'opcode e il resto un operando immediato o l'indirizzo dell'operando.

a. A quanto ammonta la massima quantità di memoria direttamente indirizzabile (in byte)?

b. Discutere l'impatto sulla velocità del sistema se il bus del microprocessore ha:

1. Un bus locale degli indirizzi a 32 bit e un bus locale dei dati a 16 bit, oppure
2. Un bus locale degli indirizzi a 16 bit e un bus locale dei dati a 16 bit.

c. Quanti bit sono necessari per il program counter e il registro delle istruzioni?

1.3 Si consideri un ipotetico microprocessore che generi indirizzi a 16 bit (ad esempio, si assuma che il program counter e i registri degli indirizzi abbiano un'ampiezza di 16 bit) e che possieda un bus dei dati a 16 bit.

a. Qual è il massimo spazio di indirizzi di memoria cui il processore può accedere direttamente se è connesso a una "memoria a 16 bit"?

b. Qual è il massimo spazio di indirizzi di memoria cui il processore può accedere direttamente se è connesso a una "memoria a 8 bit"?

c. Quali caratteristiche architetturali permetteranno a questo microprocessore di accedere ad uno "spazio di I/O" separato?

d. Se le istruzioni di input e di output possono specificare un numero di porta di I/O a 8 bit, quante porte di I/O a 8 bit possono essere supportate dal microprocessore? Quante porte di I/O a 16 bit? Motivare le risposte.

1.4 Si consideri un microprocessore a 32 bit con un bus esterno dei dati a 16 bit guidato da un clock a 8 MHz. Si assuma che questo microprocessore abbia un ciclo di bus la cui durata minima sia uguale a 4 cicli di clock. Qual è la velocità massima di trasferimento dei dati che questo microprocessore può sostenere? Per incrementare le sue prestazioni sarebbe meglio estendere il bus esterno dei dati a 32 bit, o duplicare la frequenza del clock fornita al microprocessore? Motivare le risposte e le eventuali ipotesi aggiuntive.

1.5 Si consideri un computer contenente un modulo di I/O che controlla una semplice telescrivente tastiera/stampante. I seguenti registri sono contenuti nella CPU e connessi direttamente al bus di sistema:

INPR:	registro di input, 8 bit
OUTR:	registro di output, 8 bit
FGI:	flag di input, 1 bit
FGO:	flag di output, 1 bit
IEN:	abilitazione di interrupt, 1 bit

L'input di tastiera dalla telescrivente e l'output alla stampante sono controllati dal modulo di I/O. La telescrivente è in grado di codificare un simbolo alfanumerico in una parola a 8 bit, e decodificare una parola a 8 bit in un simbolo alfanumerico. Il flag di input è settato quando una parola a 8 bit entra nel registro di input dalla telescrivente. Il flag di output è settato quando una parola è stampata.

a. Descrivere come la CPU, usando i primi 4 registri elencati in questo problema, possa gestire l'I/O con la telescrivente.

b. Descrivere come la funzione possa essere effettuata con maggior efficienza impiegando anche IEN.

1.6 Un sistema di memoria principale consiste di moduli di memoria collegati al bus di sistema dell'ampiezza di una parola. Quando viene effettuata una richiesta di scrittura, il bus è occupato per 100 ns da dati, indirizzi e segnali di controllo. Durante i medesimi 100 ns, e per i seguenti 500 ns, il modulo di memoria indirizzato esegue un ciclo, accettando i dati e memorizzandoli. Le operazioni dei moduli di memoria possono sovrapporsi, ma

solo una richiesta può essere presente sul bus ogni momento.

a. Si assuma che ci siano 8 di tali moduli connessi al bus. Qual è la massima velocità possibile (in parole/secondo) a cui i dati possono essere memorizzati?

b. Disegnare un grafico della velocità massima di scrittura come funzione del ciclo di tempo del modulo, considerando che ci sono 8 moduli di memoria e un tempo di occupazione del bus di 100 ns.

1.7 Praticamente in tutti i sistemi che includono i moduli DMA viene data maggior priorità all'accesso DMA alla memoria principale, piuttosto che all'accesso del processore alla memoria principale. Perché?

1.8 Un modulo DMA sta trasferendo caratteri nella memoria principale da un dispositivo esterno, trasmettendo 9600 bit/ secoodo (bps). Il processore può prelevare istruzioni alla frequenza di 1000000 istruzioni /secondo. Di quanto verrà rallentato il processore a causa delle operazioni DMA?

1.9 Un computer consiste di una CPU e di un dispositivo D di I/O connesso alla memoria principale M attraverso un bus condiviso con un'ampiezza dei dati di una parola. La CPU può eseguire un massimo di 10^6 istruzioni/secondo. Un'istruzione mediamente richiede 5 cicli di macchina, 3 dei quali usano il bus della memoria. Un'operazione di lettura o scrittura in memoria utilizza un ciclo di macchina. Si supponga che la CPU esegua di continuo programmi di background che richiedono il 95% del tempo di esecuzione delle sue istruzioni, ma nessuna istruzione di I/O. Si assuma che un ciclo di processore sia uguale ad un ciclo del bus. Si supponga ora che blocchi molto grandi di dati debbano essere trasferiti fra M e D.

a. Se viene utilizzato un I/O programmato, e se ciascun trasferimento I/O di una parola richiede che la CPU esegua due istruzioni, stimare la velocità massima possibile di trasferimento dei dati di I/O in parole/secondo attraverso D.

b. Stimare la medesima velocità di trasferimento, se viene utilizzato il DMA.

1.10 Estendere le equazioni (1.1) e (1.2) dell'Appendice 1A a gerarchie di memoria a n livelli.

1.11 Si consideri un sistema di memoria con i seguenti parametri:

$T_c = 100$ ns $\quad C_c = 0.01$ cent /bit

$T_m = 1200$ ns $\quad C_m = 0.001$ cent /bit

a. Qual è il costo di un megabyte di memoria principale?

b. Qual è il costo di un megabyte di memoria principale se si utilizza la memoria cache?

c. Se il tempo di accesso effettivo è più grande del 10% rispetto al tempo di accesso della cache, qual è la hit ratio H?

1.12 Un computer ha una cache, una memoria principale e un disco utilizzato per la memoria virtuale. Se si fa riferimento ad una parola che si trova nella cache, sono richiesti 20 ns

per accedervi. Se la parola si trova nella memoria principale, ma non nella cache, occorrono 60 ns per caricarvela, poi viene nuovamente referenziata. Se la parola non è nella memoria principale occorrono 12 ms per prelevarla dal disco e 60 ns per copiarla nella cache, quindi viene nuovamente referenziata. La hit ratio della cache è 0.9, quella della memoria principale è 0.6. Qual è il tempo medio in ns richiesto per accedere ad una parola referenziata in questo sistema?

1.13 Si supponga che il processore debba usare uno stack per gestire chiamate e ritorni da procedure. Si può eliminare il program counter, usando in sua vece la cima dello stack?

Appendice 1A Caratteristiche delle prestazioni delle memorie a due livelli

In questo capitolo si è fatto riferimento ad una cache che agisce come un buffer tra la memoria principale e il processore, creando una memoria interna a due livelli. Questa architettura a due livelli fornisce prestazioni superiori rispetto alla memoria ad un livello, sfruttando la proprietà nota come località, di cui tratta questa appendice.

Il meccanismo memoria principale/cache è una parte dell'architettura del computer, implementata in hardware e tipicamente invisibile al sistema operativo, e pertanto l'argomento esula dall'ambito di questo lavoro. Esistono tuttavia altri due esempi di memoria a due livelli che sfruttano anch'essi la località e che, almeno parzialmente, vengono implementati nel sistema operativo: la memoria virtuale e la cache del disco (Tabella 1.2), discussi, rispettivamente, nei Capitoli 8 e 11. Questa appendice esamina alcune delle caratteristiche delle prestazioni delle memorie a due livelli, comuni a tutti e tre gli approcci.

Località

Il principio di località, citato nella Sezione 1.5, è alla base delle migliori prestazioni delle memorie a due livelli. Esso afferma che i riferimenti alla memoria tendono a raggrupparsi: in un periodo di tempo lungo, i cluster in uso cambiano, ma nel breve periodo il processore lavora principalmente con cluster fissi di riferimenti alla memoria.

Intuitivamente, il principio di località appare sensato. Consideriamo questa serie di osservazioni:

1. Tranne che per le istruzioni di trasferimento e chiamata a subroutine, che costituiscono solo una piccola parte di tutte le istruzioni di programma, l'esecuzione del programma è sequenziale. Quindi, nella maggior parte dei casi, l'istruzione successiva da prelevare segue immediatamente l'ultima appena prelevata.
2. È raro avere una lunga sequenza ininterrotta di chiamate di procedura, seguita dalla corrispondente sequenza di ritorni da procedura. Piuttosto, il programma rimane confinato in una finestra piuttosto stretta relativamente alla profondità delle chiamate di procedura. Quindi, in un periodo di tempo breve i riferimenti alle istruzioni tendono ad essere localizzati in poche procedure.

Tabella 1.2 *Caratteristiche delle memorie a due livelli*

	Cache della memoria principale	Memoria virtuale (paginata)	Cache del disco
Rapporto tipico dei tempi di accesso	5/1	1000/1	1000/1
Sistema di gestione della memoria	Implementato da hardware speciale	Combinazione di hardware e software di sistema	Software di sistema
Dimensione tipica del blocco	Da 4 a 128 byte	Da 64 a 4096 byte	Da 64 a 4096 byte
Accesso del processore al secondo livello	Accesso diretto	Accesso indiretto	Accesso indiretto

3. La maggioranza dei costrutti iterativi consiste di un numero relativamente piccolo di istruzioni, ripetute molte volte. Per la durata dell'iterazione, la computazione è confinata in una piccola parte contigua di un programma.
4. In molti programmi buona parte del calcolo riguarda l'elaborazione di strutture dati, come array o sequenze di record, e spesso i riferimenti successivi a queste strutture dati riguarderanno gli elementi vicini.

Questa serie di osservazioni è avvalorata da numerosi studi teorici. In riferimento al punto 1, diversi autori hanno analizzato il comportamento di programmi scritti in linguaggi ad alto livello. La Tabella 1.3 riassume i principali risultati, che misurano la frequenza di vari tipi di istruzioni durante l'esecuzione, desunti dai seguenti studi. Il primo lavoro sul comportamento dei linguaggi di programmazione, dovuto a Knuth [KNUT71], esamina un insieme di programmi FORTRAN usati come esercizi per gli studenti. Tanenbaum [TANE78], ha pubblicato misura-

Tabella 1.3 *Frequenza dinamica relativa delle operazioni di linguaggi ad alto livello*

Studio	[HUCK83]	[KNUT71]	[PATT82]		[TANE78]
Linguaggio	Pascal	FORTRAN	Pascal	C	SAL
Carico di lavoro	Scientifico	Studente	Sistema	Sistema	Sistema
Assegnamento	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	-	3	-
Altro	-	7	6	1	6

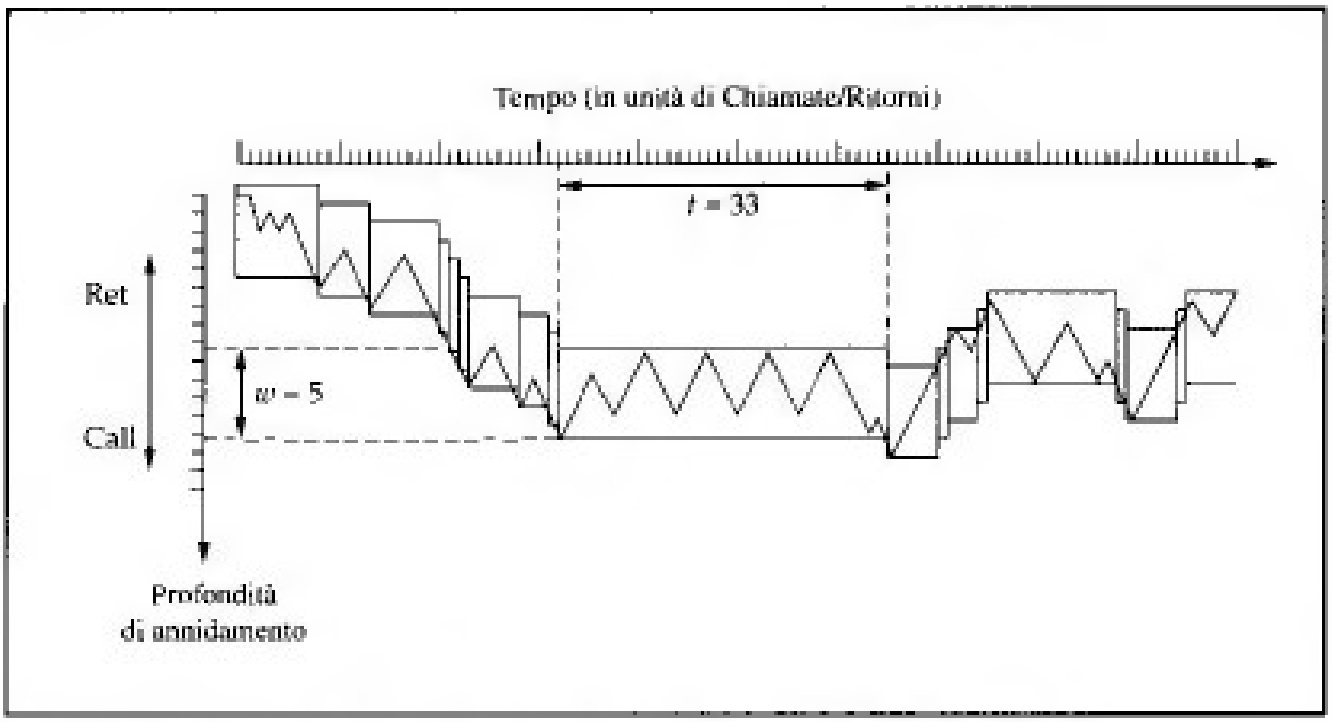


Figura 1.20 *Il comportamento Chiamata-Ritorno dei programmi*

zioni raccolte da oltre 300 procedure usate in programmi di sistemi operativi e scritte in un linguaggio che supporta la programmazione strutturata (SAL). Patterson e Sequein [PATT82] analizzano un insieme di misure ottenute da compilatori e programmi per impaginazione di testi, computer aided design (CAD), ordinamento e confronto di file. Sono stati utilizzati i linguaggi C e Pascal. Huck [HUCK83] ha analizzato quattro programmi, che rappresentano casi tipici di elaborazione scientifica general purpose, tra cui la trasformata veloce di Fourier e l'integrazione di sistemi di equazioni differenziali. I dati ricavati da questo insieme di linguaggi e applicazioni sono concordi nel mostrare che le istruzioni di trasferimento e chiamata a procedura rappresentino solo una frazione delle istruzioni eseguite durante l'esecuzione di un programma. Questi stodi, quindi, confermano la prima delle osservazioni sopraelencate.

Per quanto riguarda il secondo punto, troviamo conferma negli studi riportati in [PATT85].

La Figura 1.20 illustra il comportamento delle chiamate-ritorni da procedura. Ciascuna chiamata è rappresentata da una linea orientata in basso e a destra, e ciascun ritorno da una linea orientata in alto e a destra. Nella figura è definita una *finestra* con profondità uguale a 5. Solo una sequenza di chiamate e ritorni da procedura, con un movimento netto di 6 in una direzione qualsiasi, causa lo spostamento della finestra. Come si nota, l'esecuzione del programma può rimanere entro una finestra stazionaria per lunghi periodi di tempo. Secondo uno studio dovuto agli stessi analisti dei programmi C e Pascal, una finestra di profondità 8 dovrà spostarsi per meno dell'1% del totale delle chiamate e ritorni da procedure [TAMI83].

La validità del principio di località dei riferimenti trova peraltro continue conferme, anche in ricerche più recenti. Ad esempio, la Figura 1.21 illustra i risultati di uno studio sui pattern di accesso a pagine Web in un singolo sito.

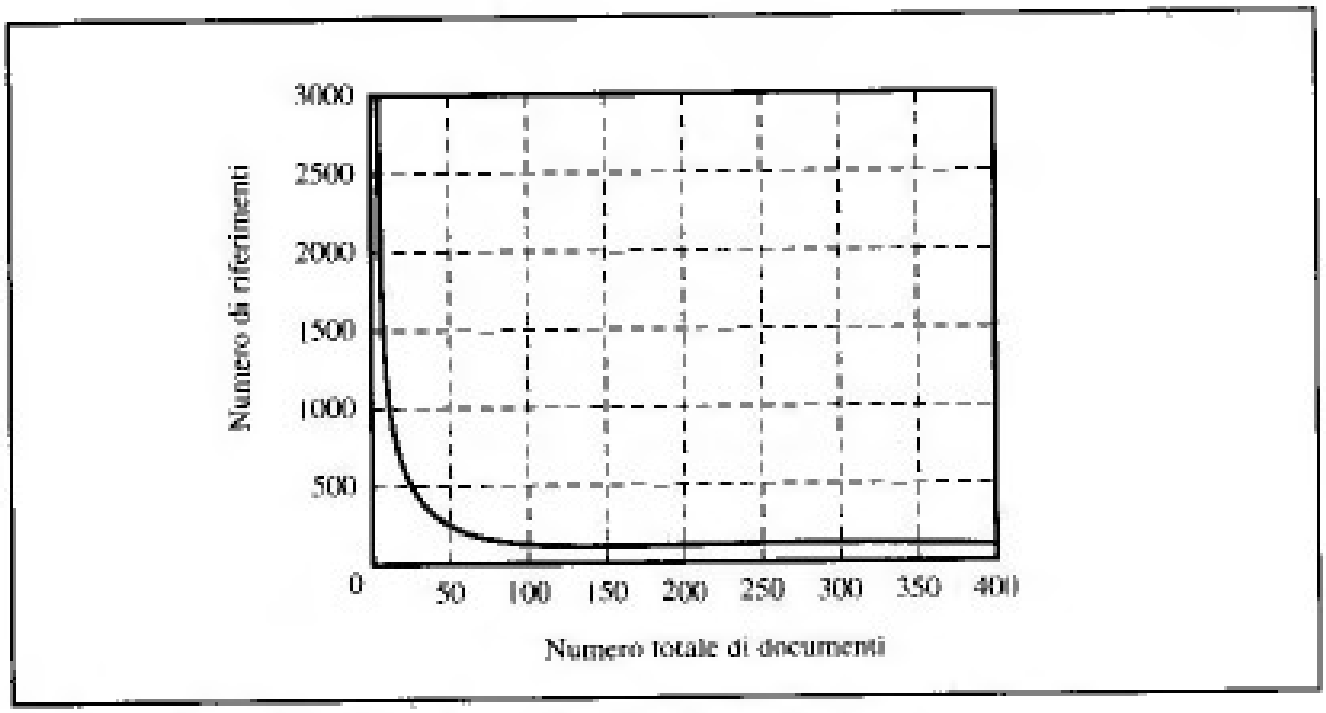


Figura 1.21 *Località di riferimento per le pagine Web [BAEN97]*

Operazioni della memoria a due livelli

È possibile sfruttare la proprietà di località nella costruzione di una memoria a due livelli. La memoria di livello superiore (M1) è più piccola, più veloce e più costosa (per bit) di quella di livello più basso (M2). M1 viene utilizzata per memorizzare temporaneamente una parte del contenuto di M2. Quando si fa un riferimento alla memoria, viene effettuato un tentativo di accesso a M1. Se riesce, si ha un accesso veloce, altrimenti un blocco di locazioni di memoria viene copiato da M2 a M1 e si ha un accesso via M1. A causa della località, una volta che il blocco è trasferito in M1, si dovrebbe avere un buon numero di accessi a locazioni in quel blocco, con il risultato di un servizio complessivo più veloce.

Per esprimere il tempo medio di accesso ad un elemento, occorre considerare non solo le velocità dei due livelli di memoria, ma anche la probabilità che un certo riferimento a M1 abbia successo. Si ha

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{1.1}$$

Dove

T_s = tempo di accesso medio del sistema

T_1 = tempo di acceso di M1 (cioè memoria cache, cache del disco)

T_2 = tempo di acceso di M2 (cioè memoria principale, disco)

H = hit ratio (probabilità dei riferimenti diretti a M1)

La Figura 1.15 mostra il tempo d'accesso medio come una funzione della hit ratio. Come si può vedere, per un'alta percentuale di hit il tempo medio di accesso è molto più vicino a quello di M1 che a quello di M2.

Prestazioni

Consideriamo alcuni parametri rilevanti per l'impostazione di un meccanismo di memoria a due livelli; e in primo luogo, il costo. Si ha

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{1.2}$$

Dove

C_s = costo medio per bit della memoria combinata a due livelli

C_1 = costo medio per bit della memoria di livello superiore M1

C_2 = costo medio per bit della memoria di livello inferiore M2

S_1 = dimensione di M1

S_2 = dimensione di M2

Sarebbe desiderabile che $C_s \approx C_2$. Dato che $C_1 >> C_2$, ciò richiede che $S_1 << S_2$. La Figura 1.22 illustra questa relazione.

Consideriamo ora il tempo di accesso. Per ottenere un miglioramento rilevante nelle prestazioni di una memoria a due livelli, è necessario che T_s sia approssimativamente uguale a T_1 ($T_s \approx T_1$). Dato che T_1 è molto minore di T_2 ($T_1 << T_2$), è necessaria una hit ratio vicina ad 1.

Così, sarebbe opportuno che M1 fosse piccola, per mantenere bassi i costi, ma anche grande, per migliorare la hit ratio e quindi le prestazioni. Esiste una dimensione di M1 che soddisfi entrambe le esigenze in misura ragionevole? Un quesito al quale si può rispondere, solo formulando un'ulteriore serie di domande:

- Quale valore della hit ratio occorre per soddisfare i vincoli relativi alle prestazioni?
- Quale dimensione di M1 assicurerà la hit ratio necessaria?
- Questa dimensione rispetterà i vincoli di costo?

Per rispondere, si consideri la quantità T_1/T_s, definita *efficienza d'accesso*: essa misura di quanto il tempo medio d'accesso (T_s) sia vicino al tempo d'accesso a M1 (T_1). Dall'equazione (1.1) si ottiene:

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \tag{1.3}$$

La Figura 1.23 rappresenta T_1/T_s come una funzione della hit ratio H, con la quantità T_2/T_1 come parametro. Tipicamente, il tempo di accesso alla cache è circa da 5 a 10 volte inferiore a

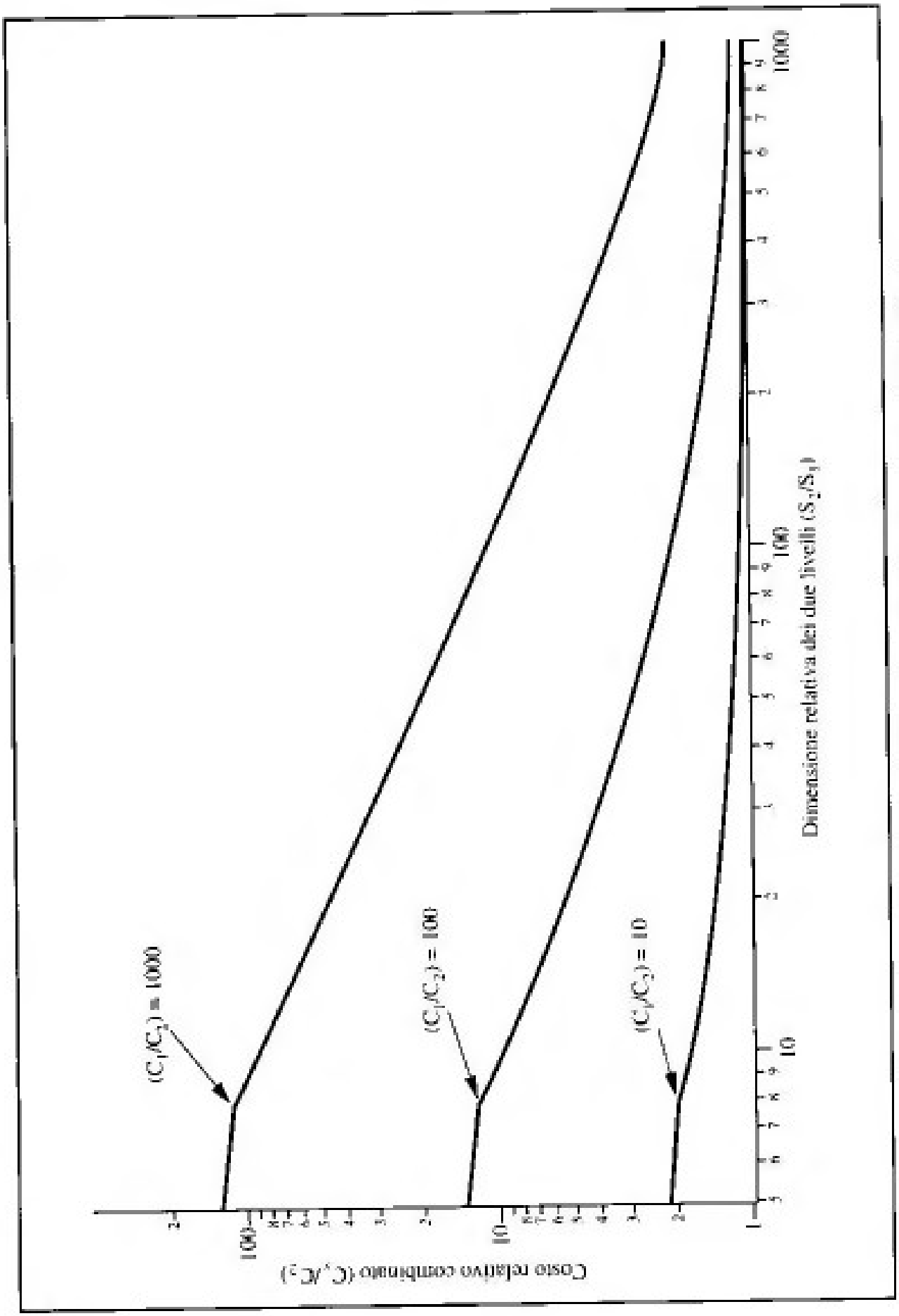


Figura 1.22 *Relazione fra il costo medio e la dimensione relativa della memoria per una memoria a due livelli*

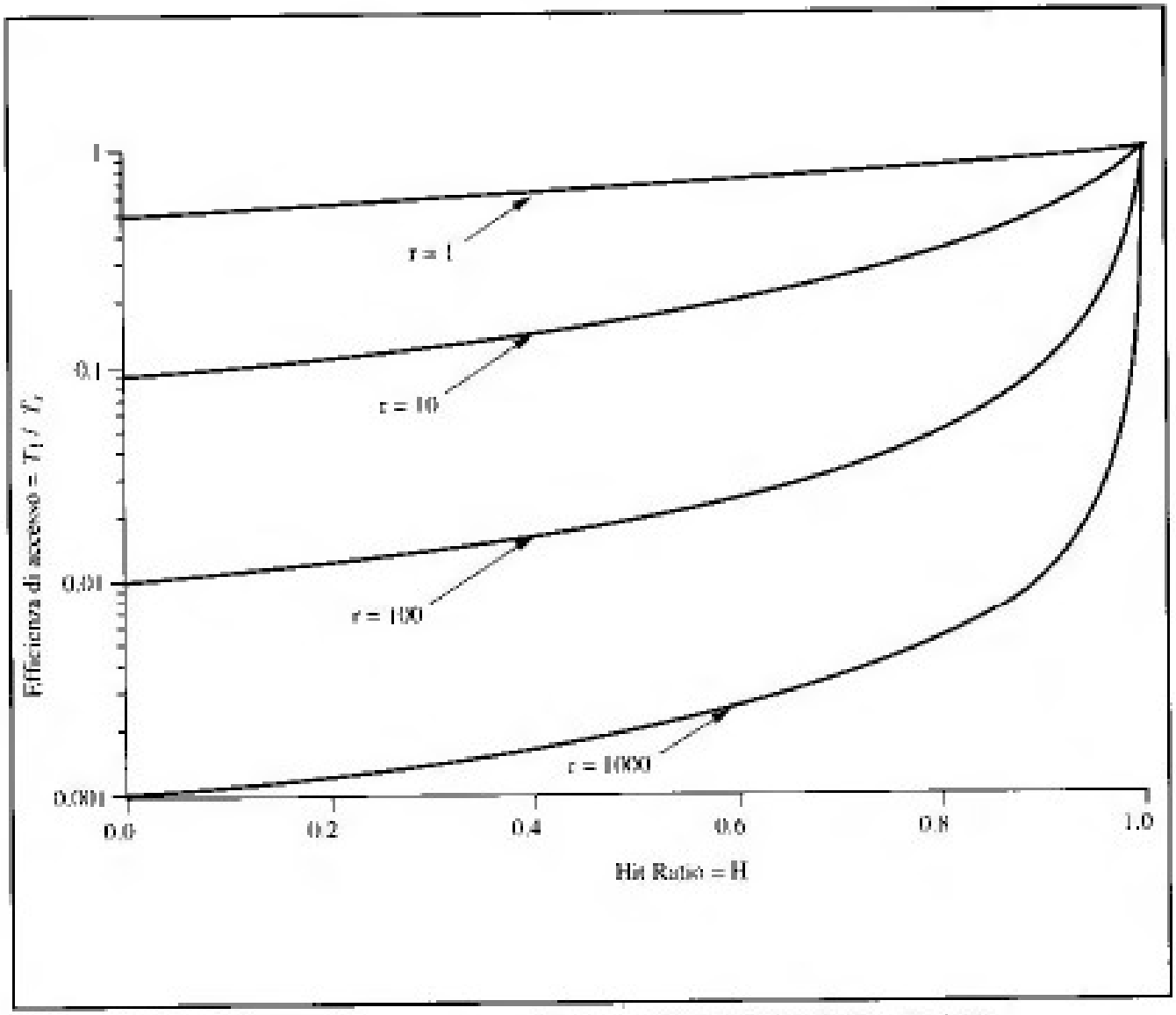


Figura 1.23 *Efficienza di accesso come funzione della Hit Ratio ($r = T_2 / T_1$)*

quello di accesso alla memoria principale (cioè T_2 / T_1 varia da 5 a 10), mentre il tempo di accesso alla memoria è circa 1000 volte inferiore a quello di accesso al disco ($T_2 / T_1 = 1000$). Quindi, per soddisfare i vincoli delle prestazioni, sembrerebbe necessaria una hit ratio compresa fra 0.8 e 0.9.

A questo punto, è possibile formulare con maggior precisione la domanda inerente la dimensione della memoria. È ragionevole una hit ratio di 0.8 o migliore per $S_1 << S_2$? La risposta dipende da diversi fattori, tra cui la natura del software in esecuzione e le caratteristiche dell'architettura della memoria a due livelli. L'elemento decisivo naturalmente è il grado di località. La Figura 1.24 chiarisce l'effetto della località sulla hit ratio. Evidentemente, se M1 ha la stessa dimensione di M2, la hit ratio sarà 1.0: tutti gli elementi in M2 sono sempre memorizzati anche in M1. Supponiamo ora che, in assenza di località, i riferimenti siano completamente casuali. In tal caso, la hit ratio sarebbe una funzione strettamente lineare della dimensione relativa della memoria. Ad esempio, se la dimensione di M1 fosse metà di quella di M2, in ogni momento metà degli elementi di M2 sarebbero anche in M1, con hit ratio di 0.5. In pratica, tuttavia, esiste

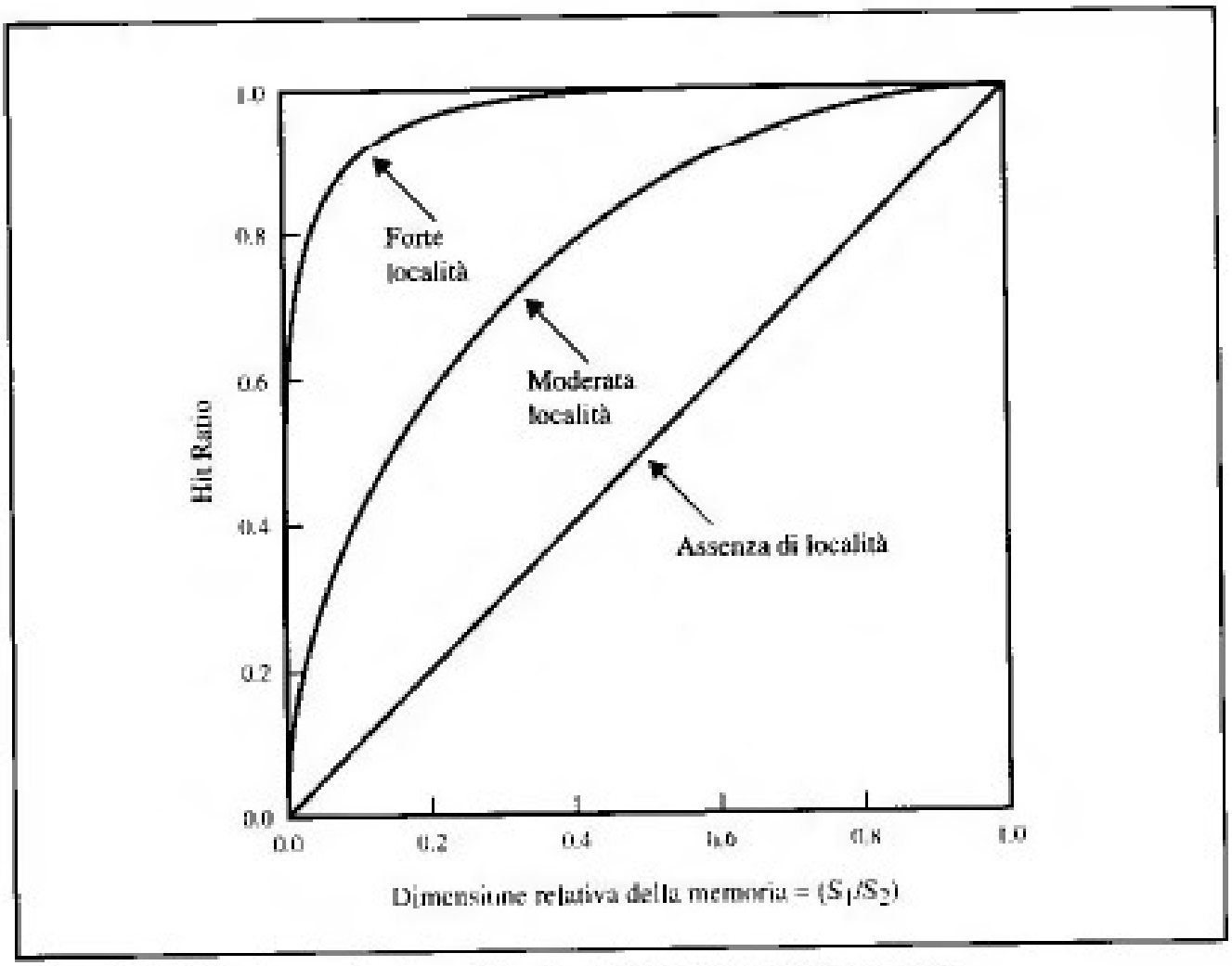


Figura 1.24 *Hit Ratio come funzione della dimensione relativa della memoria*

un certo grado di località nei riferimenti. La figura rappresenta gli effetti della località, nei casi in cui sia moderata oppure forte.

Pertanto, se c'è forte località, è possibile ottenere alti valori della hit ratio anche con dimensioni relativamente ridotte della memoria di livello superiore. Ad esempio, numerosi studi hanno dimostrato che dimensioni piuttosto ridotte della cache conducono a hit ratio superiori a 0.75, *indipendentemente dalla dimensione della memoria principale* ([AGAR89a], [AGAR89b], [PRZY88], [STRE83] e [SMIT82]). Una cache compresa tra 1K e 128K word (*parole*) è generalmente adeguata, mentre la memoria principale è solitamente dell'ordine di diversi megabyte. Esaminando più avanti la memoria virtuale e la cache del disco, citeremo altri studi a conferma del fatto che una M1 relativamente piccola conduce ad alti valori della hit ratio a causa della località.

Ci ritroviamo finalmente all'ultima delle domande formulate in precedenza: la dimensione relativa delle due memorie soddisfa i vincoli di costo? La risposta è senz'altro affermativa: se abbiamo bisogno soltanto di una memoria di livello superiore relativamente piccola per ottenere buone prestazioni, allora il costo medio per bit dei due livelli di memoria si avvicinerà a quello della memoria di livello inferiore, meno cara.

Appendice 1B Gestione delle procedure

Per controllare l'esecuzione delle chiamate e ritorni da procedura si ricorre comunemente all'uso di uno stack. Questa appendice riassume le proprietà fondamentali degli stack e il loro impiego nella gestione delle procedure.

Implementazione dello stack

Uno stack è un insieme ordinato di elementi cui si può accedere uno solo alla volta. Il punto di accesso è chiamato *cima* dello stack. Il numero di elementi che lo compongono, o lunghezza, è variabile. Gli elementi possono essere aggiunti o cancellati solo dalla cima dello stack. Per questa ragione, uno stack è anche chiamato lista *pushdown* o lista *last-in-first-out* (LIFO).

Per implementare uno stack occorre un insieme di locazioni usate per memorizzare i suoi elementi. La Figura 1.25 illustra un caso tipico. Nella memoria centrale (oppure in quella virtuale) si riserva per lo stack un blocco di locazioni contigue. Nella maggior parte dei casi, il blocco è parzialmente riempito con elementi dello stack, mentre il resto si può utilizzare per aggiungerne altri. Per le operazioni proprie dello stack, sono necessari tre indirizzi, spesso memorizzati nei registri del processore:

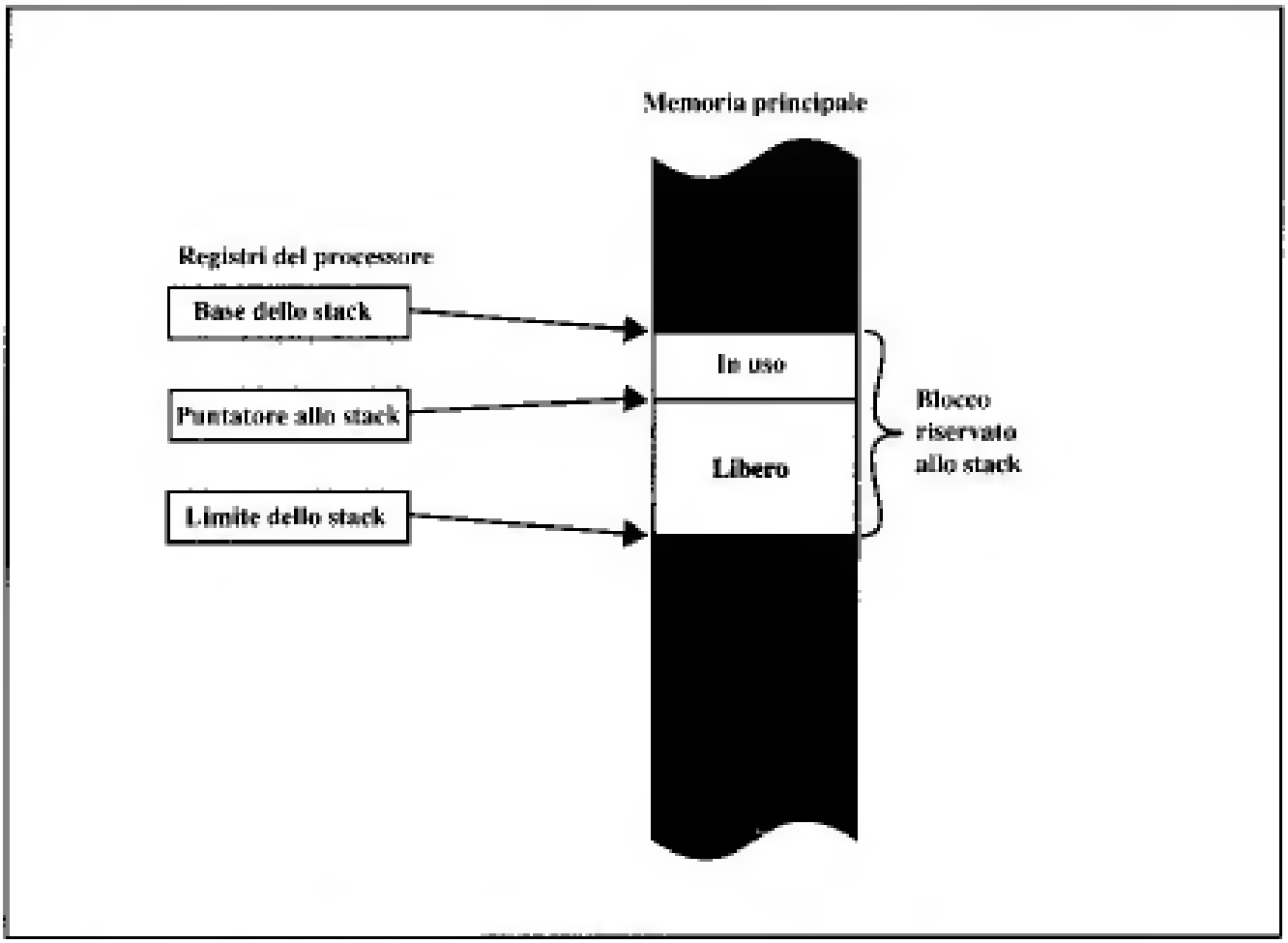


Figura 1.25 *Organizzazione tipica dello stack*

- **Stack pointer**: (*puntatore allo stack*) contiene l'indirizzo della cima dello stack. Se un elemento è aggiunto (PUSH) o cancellato (POP) dallo stack, il puntatore è incrementato o decrementato per contenere l'indirizzo della nuova cima dello stack.
- **Stack base** (*base dello stack*): contiene l'indirizzo del fondo dello stack nel blocco riservato. Questa è la prima locazione ad essere utilizzata quando un elemento è aggiunto ad uno stack vuoto. Se si tenta di effettuare un'operazione di POP quando lo stack è vuoto, si verifica un errore.
- **Stack limit** (*limite dello stack*): contiene l'indirizzo dell'altro estremo (cima o *top*) del blocco riservato. Se si tenta di effettuare un'operazione di PUSH quando lo stack è pieno, si verifica un errore.

Chiamate e ritorni da procedura

Comunemente, per gestire le chiamate e i ritorni da procedura si fa uso di uno stack. Quando il processore esegue una chiamata, salva l'indirizzo di ritorno sullo stack e, quando una procedura termina, recupera l'indirizzo dalla cima dello stack. La Figura 1.26 illustra l'uso di uno stack per gestire le procedure annidate della Figura 1.27.

Spesso una chiamata di procedura deve anche passare parametri, e potrebbe farlo per mezzo dei registri. Un'altra possibilità è immagazzinare i parametri in memoria subito dopo l'istruzione di chiamata: in questo caso, il ritorno dev'essere fissato nella locazione che segue i parametri. Entrambi gli approcci presentano degli inconvenienti. Se si usano i registri, il programma chiamato e quello chiamante devono essere scritti in modo da farlo nel modo opportuno. Mettere i parametri in memoria rende difficoltoso scambiarsi un numero variabile di parametri.

Un approccio più flessibile consiste nel passare i parametri attraverso lo stack. Quando il processore esegue una chiamata, non salva sullo stack solo l'indirizzo di ritorno, ma anche i parametri da passare alla procedura chiamata, che può accedere ad essi tramite lo stack. Alla fine della procedura, i parametri di ritorno possono ancora essere salvati sullo stack, *sotto* l'indirizzo di ritorno. L'intero insieme dei parametri, unitamente all'indirizzo di ritorno, memorizzati per una chiamata di procedura, sono definiti come **stack frame**.

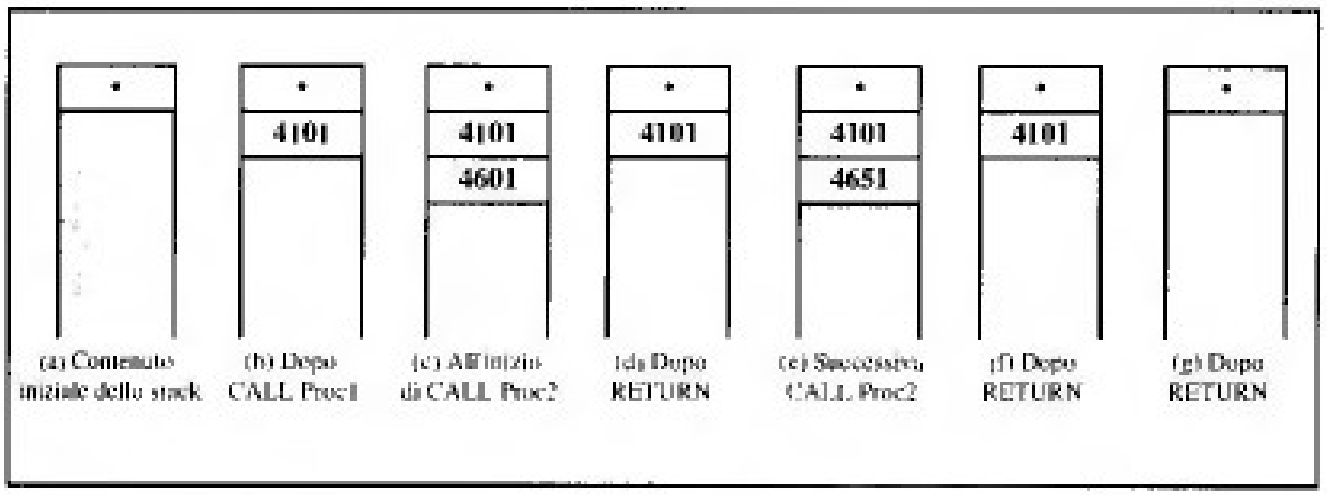


Figura 1.26 *Uso dello stack per implementare le procedure annidate della Figura 1.27*

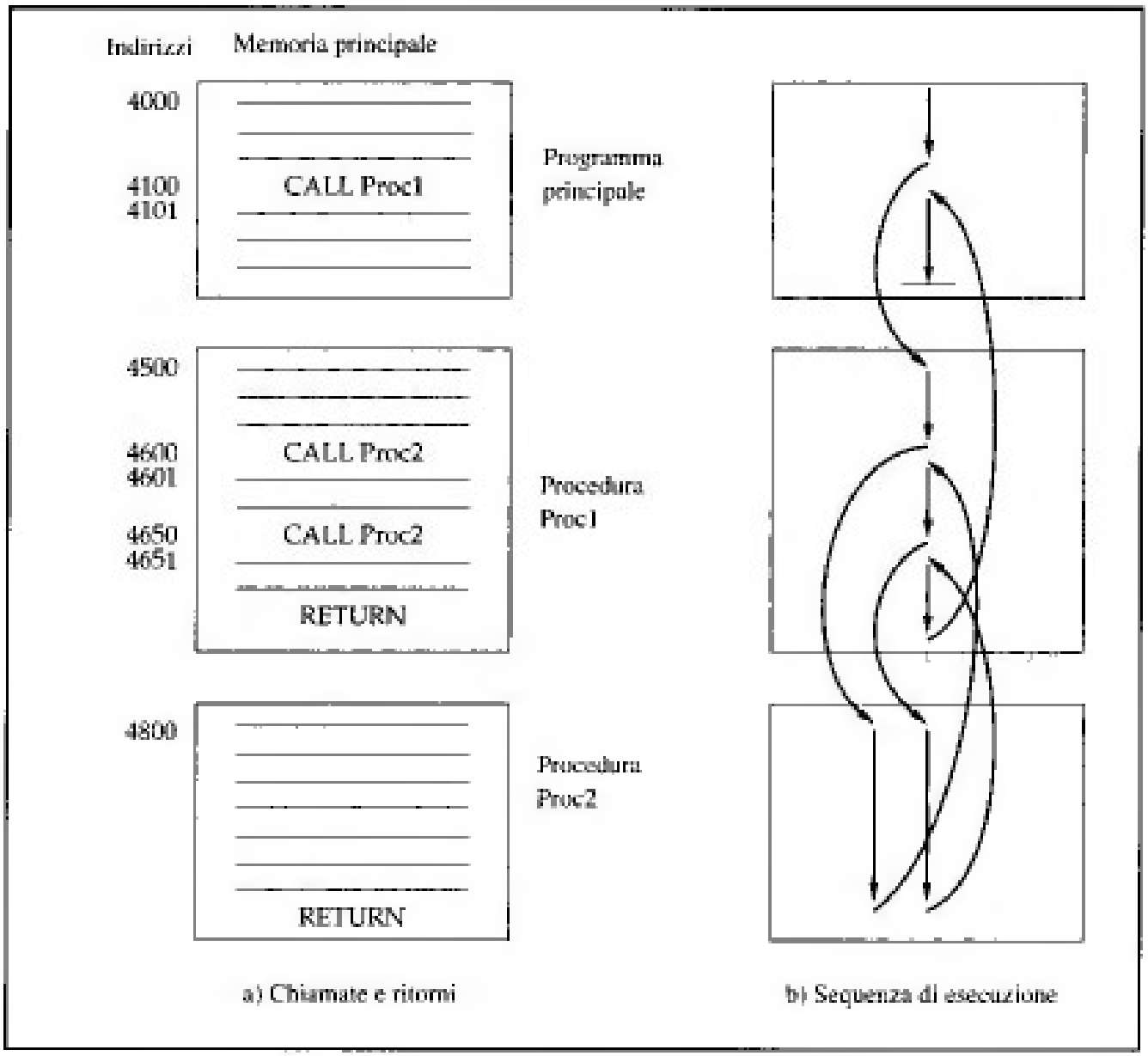


Figura 1.27 *Procedure annidate*

La Figura 1.28 presenta un esempio: esso si riferisce alla procedura *P*, in cui sono dichiarate le variabili locali *x*1 e *x*2, e alla procedura *Q*, in cui sono dichiarate le variabili locali *y*1 e *y*2, e che può essere chiamata da *P*. In questa figura, il punto di ritorno per ciascuna procedura è il primo elemento memorizzato nel corrispondente stack frame. Successivamente, è memorizzato un puntatore all'inizio del frame precedente, operazione necessaria, se il numero o la lunghezza dei parametri da mettere nello stack sono variabili.

Procedure rientranti

Un concetto utile, in particolare in un sistema multiutente, è quello di procedura rientrante. Essa è caratterizzata dal fatto che una singola copia del codice del programma può essere condivisa da più utenti contemporaneamente. Le procedure rientranti hanno due caratteristiche chia-

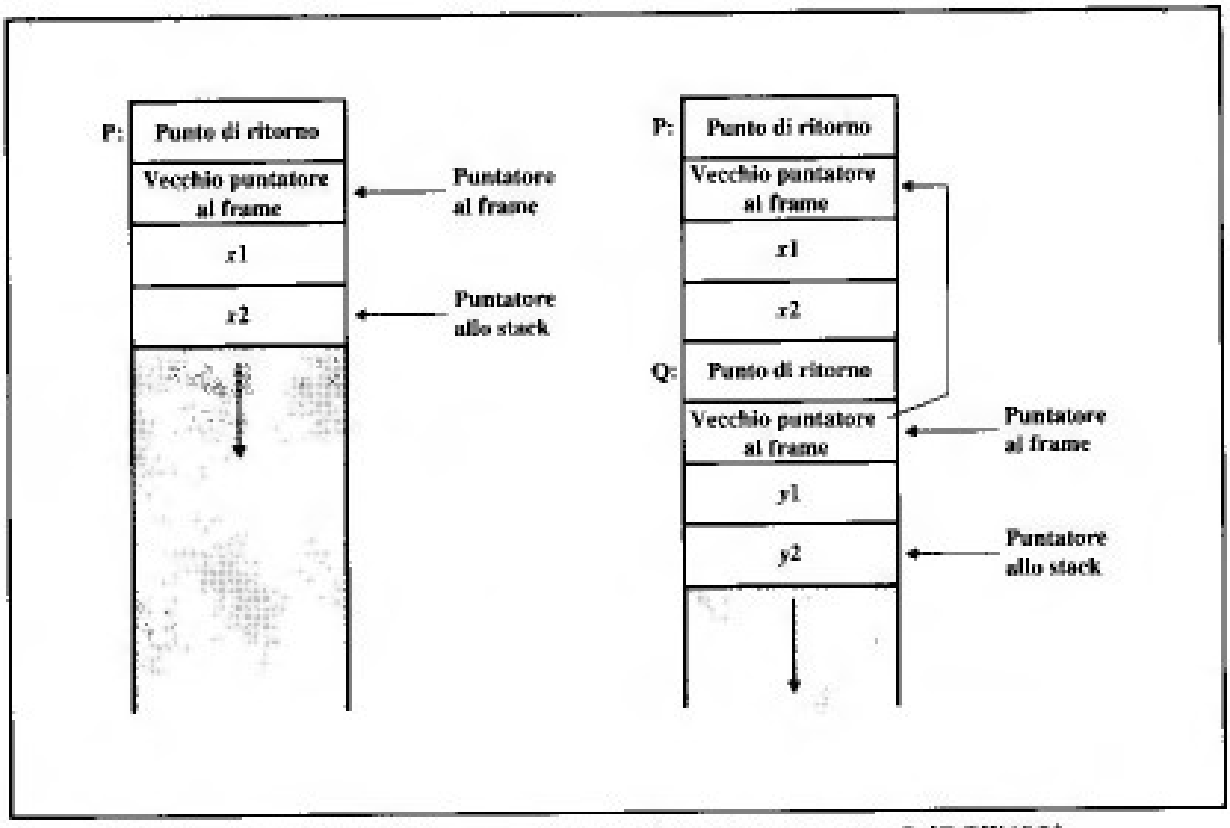


Figura 1.28 *Crescita dello stack frame per le procedure di esempio P e Q [DEWA90]*

ve: il codice del programma non può automodificarsi, e i dati locali relativi a ciascun utente devono essere memorizzati separatamente. Esse possono essere interrotte e chiamate da un programma, e possono ancora continuare correttamente l'esecuzione al ritorno dalla procedura. In un sistema condiviso, le procedure rientranti consentono di usare in modo più efficiente la memoria principale: una copia del codice del programma è mantenuta nella memoria principale, ma più di un'applicazione può chiamare la procedura.

Quindi una procedura rientrante deve avere una parte permanente (le istruzioni che costituiscono la procedura) e una temporanea (un puntatore al programma chiamante, e della memoria per le variabili locali usate dal programma). Ciascuna istanza di esecuzione di una procedura, detta *attivazione*, eseguirà il codice nella parte permanente, ma deve avere la sua copia di variabili locali e parametri. La parte temporanea, associata ad una particolare attivazione, è definita *record di attivazione*.

Le procedure rientranti sono implementate al meglio per mezzo di uno stack: quando una di queste procedure è chiamata, il suo record di attivazione può essere memorizzato sullo stack, divenendo quindi parte dello stack frame, creato alla chiamata della procedura.

C A P I T O L O 2

INTRODUZIONE AI SISTEMI OPERATIVI

Questo capitolo traccia una breve storia dei sistemi operativi, interessante in se ma soprattutto utile per fornire una panoramica sui loro principi costitutivi. Dopo un rapido sguardo sui loro obiettivi e le loro funzioni, se ne prende in esame l'evoluzione, dai primi sistemi batch fino a quelli multiutente e multimodo, assai più raffinati. Il resto del capitolo è dedicato alla storia e alle caratteristiche generali dei due sistemi operativi adottati come esempio in questo testo.

2.1 Obiettivi e funzioni dei sistemi operativi

Un sistema operativo è un programma che controlla l'esecuzione di programmi applicativi e agisce come un'interfaccia fra l'utente e l'hardware del computer, avendo 3 obiettivi di fondo:

- **Convenienza:** un sistema operativo è ciò rende conveniente l'uso di un computer
- **Efficienza:** un sistema operativo permette di utilizzare le risorse del sistema di elaborazione in modo efficiente.
- **Capacità di evolversi:** un sistema operativo dovrebbe essere costruito in modo da permettere lo sviluppo, il testing e l'introduzione di nuove funzioni di sistema senza interferire con l'attività del sistema operativo stesso.

Passiamo ora ad esaminare questi tre aspetti, uno per volta.

Il sistema operativo come interfaccia utente/computer

L'hardware e il software utilizzati per fornire applicazioni all'utente possono essere strutturati gerarchicamente, come illustrato dalla Figura 2.1. L'utente finale delle applicazioni, in genere, non si preoccupa dell'architettura del computer: vede il sistema di elaborazione come un'applicazione espressa in un linguaggio di programmazione e già sviluppata da un programmatore. Dovendo sviluppare un programma applicativo come un insieme di istruzioni macchina, completamente responsabile del controllo dell'hardware del computer, ci si troverebbe di fronte un compito assai complesso. Al fine di renderlo più agevole, è fornito un insieme di programmi di sistema, alcuni dei quali sono chiamati utility, che implementano funzioni di uso ricorrente, come assistere nella creazione dei programmi, nella gestione dei file e nel controllo dei dispositivi di I/O. Il programmatore farà uso di queste funzioni di utilità (*facility*) nello sviluppo di un'applicazione che, nel corso dell'esecuzione, chiamerà le utility per effettuare determinate funzioni. Il sistema operativo, che è il programma di sistema più importante, nasconde i dettagli dell'hardware al programmatore e gli fornisce un'interfaccia conveniente per usare il sistema; esso agisce come un mediatore, rendendo più semplice, sia per il programmatore sia per i programmi applicativi, l'accesso e l'uso di queste funzioni di utilità e dei servizi.

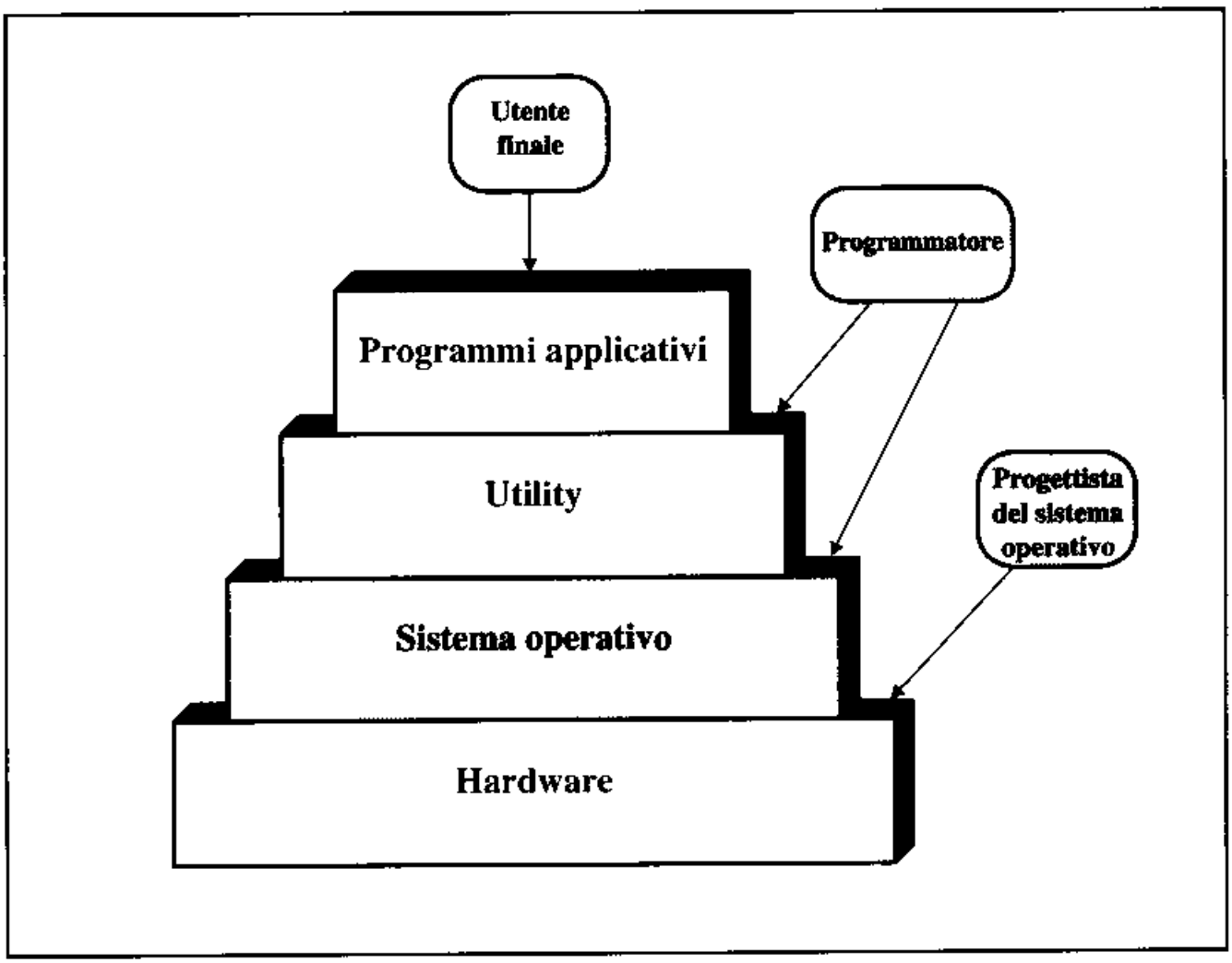


Figura 2.1 *Strati e viste di un sistema di elaborazione*

In breve, questi sono i servizi tipicamente fornitì da un sistema operativo:

- **Creazione dei programmi**: per assistere il programmatore, il sistema operativo fornisce diverse funzioni di utilità e servizi, come l'editor e il debugger, in forma di programmi di utilità che non fanno parte nel sistema operativo, ma sono accessibili attraverso di esso.
- **Esecuzione dei programmi**: per eseguire un programma, è necessario che siano compiute diverse azioni, dal caricamento nella memoria principale di istruzioni e dati, all'inizializzazione dei dispositivi di I/O, alla preparazione delle altre risorse. Il sistema operativo gestisce tutto questo per l'utente.
- **Accesso ai dispositivi di I/O**: ognuno di essi opera attraverso un proprio insieme particolare di istruzioni o segnali di controllo. Il sistema operativo si occupa dei dettagli, in modo che il programmatore possa pensare in termini di semplici operazioni di lettura e scrittura.
- **Accesso controllato ai file**: nel caso dei file il controllo comprende non solo la comprensione della natura del dispositivo di I/O (drive del disco o del nastro), ma anche quella del formato del file sul supporto di memorizzazione. È ancora il sistema operativo a occuparsi dei dettagli; inoltre, nei sistemi con molti utenti contemporanei, può fornire meccanismi di protezione per controllare l'accesso ai file.
- **Accesso al sistema**: nel caso di sistemi condivisi o pubblicamente accessibili, il sistema operativo controlla l'accesso al sistema nel suo complesso, nonché alle singole risorse. La funzione di accesso protegge dati e risorse dagli utenti non autorizzati, e risolve i conflitti per l'accesso alle risorse stesse.
- **Rilevazione degli errori e risposta**: mentre un sistema di elaborazione sta funzionando, possono verificarsi molti errori, sia hardware, interni ed esterni (ad esempio, un errore di memoria, un guasto o il malfunzionamento di un dispositivo), sia software (un overflow aritmetico, un tentativo di accedere a locazioni di memoria vietate e l'impossibilità da parte del sistema operativo di garantire il buon esito della richiesta di un'applicazione). In tali casi, il sistema operativo deve fornire una risposta che elimini la condizione di errore con il minore impatto possibile sulle applicazioni in esecuzione, terminando il programma che ha causato l'errore, ritentando l'operazione o semplicemente comunicando l'errore all'applicazione.
- **Contabilità**: un buon sistema operativo raccoglie statistiche d'uso delle diverse risorse e tiene sotto controllo i vari parametri di prestazione, tra cui il tempo di risposta. In ogni sistema ciò consente di prevenire la necessità di fare miglioramenti e di regolare il sistema al fine di migliorare le prestazioni. In sistemi multiutente, questo genere di informazioni è utilizzato anche per la fatturazione di eventuali costi da addebitare agli utenti.

Sistema operativo come gestore delle risorse

Un computer costituisce un insieme di risorse finalizzate a trasferire dati, memorizzarli ed elaborarli, nonché al controllo di queste stesse funzioni. Responsabile della gestione di tali risorse è il sistema operativo.

È possibile affermare che sia il sistema operativo a controllare il trasferimento, la

memorizzazione e l'elaborazione dei dati? In un certo senso, la risposta è affermativa: poiché gestisce le risorse del computer, il sistema operativo ne controlla le funzioni base, in un modo però piuttosto singolare. Normalmente, un meccanismo di controllo è qualcosa di esterno, o almeno di distinto e separato da ciò che viene controllato: basti pensare ad un impianto di riscaldamento domestico, controllato da un termostato completamente distinto dagli elementi che generano e distribuiscono il calore. Così non accade nel caso del sistema operativo che, in quanto meccanismo di controllo, è insolito per due aspetti:

- Il sistema operativo funziona come il software normale del computer, essendo anch'esso un programma eseguito dal processore.
- Il sistema operativo spesso cede il controllo e deve dipendere dal processore per riaverlo.

Il sistema operativo, infatti, è un programma per computer che, al pari di tutti gli altri, fornisce istruzioni al processore, mentre le sue finalità sono sostanzialmente diverse: il sistema operativo dirige il processore nell'utilizzo delle altre risorse di sistema e nella temporizzazione dell'esecuzione degli altri programmi. Ma affinché il processore faccia questo, deve smettere di eseguire il programma del sistema operativo ed eseguirne altri. Quindi il sistema operativo cede il controllo perché il processore faccia del lavoro utile, e quindi riprende il controllo quel tanto che basta a preparare il processore ad eseguire il lavoro successivo. Procedendo nella lettura di questo capitolo, dovrebbe risultare più chiaro il meccanismo alla base di ciò.

La Figura 2.2 rappresenta le principali risorse gestite dal sistema operativo, una porzione del quale si trova nella memoria principale: essa comprende il **kernel** o **nucleo,** che contiene le funzioni del sistema operativo usate più di frequente e, a un tempo dato, altre porzioni del sistema correntemente in uso. La parte restante della memoria principale contiene altri programmi e dati dell'utente. Come vedremo, l'allocazione di questa risorsa (memoria principale) è controllata congiuntamente dal sistema operativo e dall'hardware del processore dedicato alla gestione della memoria. Il sistema operativo decide quando un programma in esecuzione possa usare un dispositivo di I/O e controlla l'accesso e l'uso dei file. Il processore stesso è una risorsa, e tocca al sistema operativo determinare quanto del tempo di processore debba essere dedicato ad un particolare programma utente, decisione che, nel caso di un sistema multiprocessore deve riguardare tutti i processori.

Facilità di evoluzione di un sistema operativo

Un sistema operativo evolve nel tempo grazie ad un certo numero di fattori:

- **Aggiornamento e nuovi tipi di hardware**: ad esempio, le prime versioni di UNIX e OS/2 non impiegavano il meccanismo di paginazione perché utilizzate su macchine prive di paginazione hardware[1], mentre le versioni più recenti sono state modificate per sfruttarne le possibilità. Inoltre, l'adozione di terminali grafici e a modalità di pagina invece di quelli a linea, può influire sulla progettazione del sistema operativo. Un terminale grafico di questo tipo permette, infatti, all'utente di vedere diverse applicazioni nello stesso tempo, usando

[1] La paginazione è introdotta brevemente più avanti in questo capitolo, è discussa in dettaglio nel Capitolo 7.

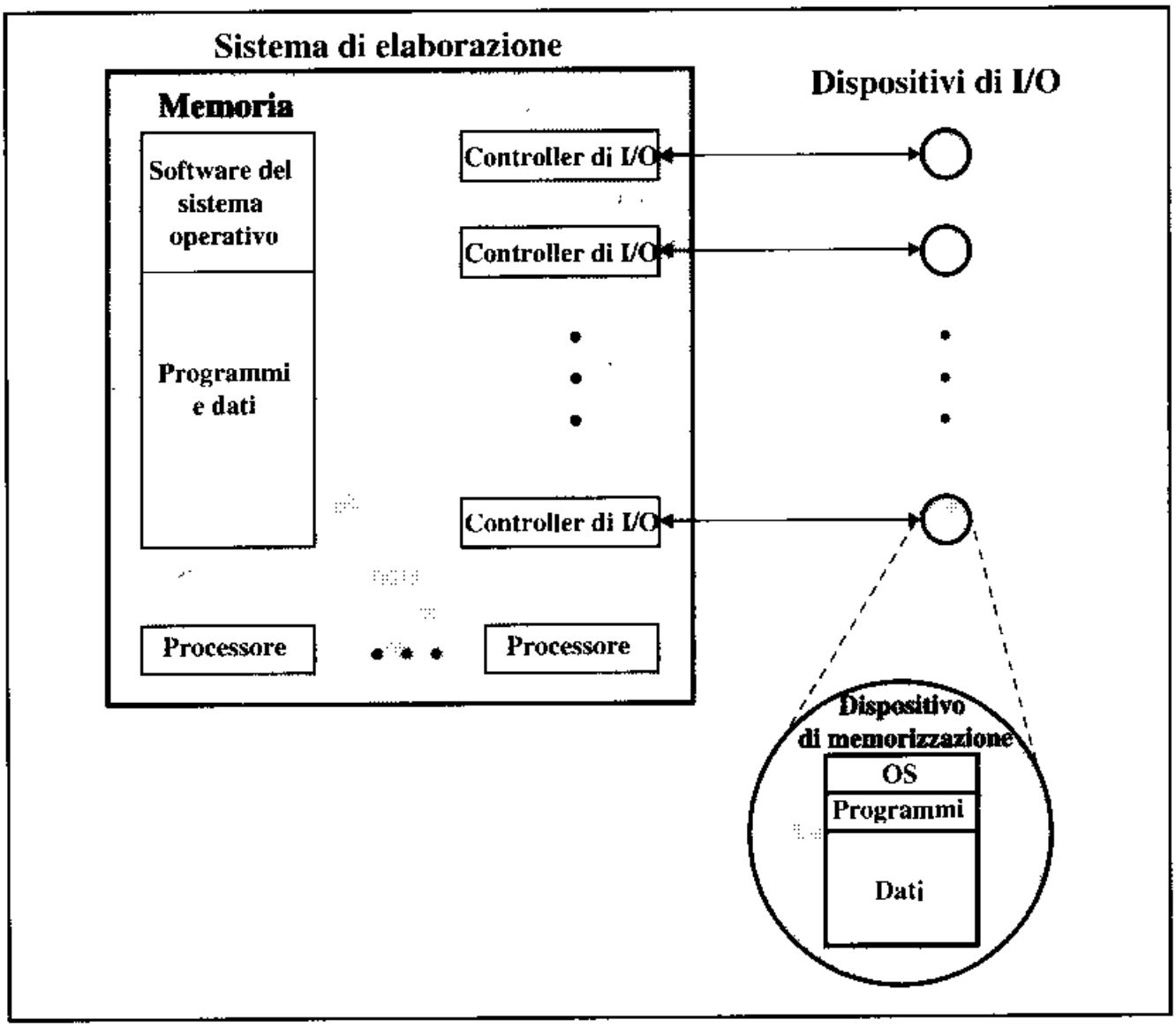


Figura 2.2 *Sistema operativo come gestore delle risorse*

delle "finestre" sullo schermo, ma richiede al sistema operativo un'azione di supporto più sofisticata.

- **Nuovi servizi**: per soddisfare le richieste degli utenti e rispondere alle necessità degli amministratori di sistema, il sistema operativo si espande per offrire nuovi servizi. Ad esempio, nel caso in cui sia difficile, con gli strumenti esistenti, mantenere buone prestazioni per gli utenti, è possibile aggiungere al sistema operativo nuovi strumenti per il controllo e la misurazione delle prestazioni. Un altro caso che ha richiesto aggiornamenti significativi del sistema è stato quello delle nuove applicazioni che utilizzano le finestre sullo schermo.
- **Correzioni**: ciascun sistema operativo è soggetto a errori e difetti, individuati nel corso del tempo e di volta in volta fatti oggetto di correzioni che, naturalmente, possono introdurre nuovi errori.

La necessità di modificare più volte nel tempo un sistema operativo, impone determinati requisiti alla sua progettazione. Ovviamente, il sistema dev'essere modulare, dotato di interfacce

fra i moduli chiaramente definite e ben documentate. Per programmi molto complessi, quali ad esempio i sistemi operativi di oggi, procedimenti definibili come una semplice modularizzazione appaiono ormai inadeguati[DENN80a]: limitarsi a suddividere un programma in subroutine non è più sufficiente. Torneremo ancora sull'argomento più avanti, nel corso di questo capitolo.

2.2 Evoluzione dei sistemi operativi

Al fine di comprendere le necessità principali dei sistemi operativi e l'importanza delle caratteristiche dei sistemi moderni, è utile esaminare come questi sistemi si siano evoluti nel corso del tempo.

Elaborazione seriale

Sui primi computer, dalla fine degli anni '40 alla metà degli anni '50, non esisteva un sistema operativo; il programmatore interagiva direttamente con l'hardware. Le macchine erano controllate da una console costituita da indicatori luminosi, interruttori a levetta, qualche forma di dispositivo di input e una stampante. I programmi in codice macchina erano caricati attraverso il dispositivo di input (ad esempio, un lettore di schede); se un errore fermava il programma, la condizione di errore veniva segnalata dagli indicatori luminosi e, per determinarne la causa, il programmatore esaminava i registri e la memoria principale. Se il programma continuava fino alla conclusione normale, l'output appariva sulla stampante.

Questi primi sistemi presentavano due problemi principali:

- **Schedulazione**: la maggior parte delle installazioni utilizzava un foglio per prenotare il tempo macchina. Generalmente un utente poteva prenotare un blocco di tempo in multipli di mezz'ora, ad esempio un'ora, e se finiva in 45 minuti, c'era uno spreco di tempo macchina. Al contrario, potevano verificarsi inconvenienti, perciò all'utente non bastava il tempo riservatogli ed era costretto a fermarsi prima di risolvere il suo problema.
- **Tempo di preparazione**: un singolo programma, chiamato **job**, poteva richiedere il caricamento del compilatore e del programma in linguaggio di alto livello (programma sorgente) nella memoria, il salvataggio del programma compilato (programma oggetto), quindi il caricamento e il link del programma oggetto e delle funzioni comuni. Ciascuno di questi passi poteva comportare il montaggio e smontaggio di nastri, oppure la preparazione di pacchi di schede. In caso di errore, lo sfortunato utente doveva tornare all'inizio della sequenza di preparazione: una quantità di tempo considerevole veniva impiegata nel preparare il programma per l'esecuzione.

Si potrebbe definire questo modo di operare elaborazione seriale, in quanto gli utenti hanno accesso al computer uno dopo l'altro in serie. Nel corso del tempo, con l'intento di rendere più efficiente l'elaborazione seriale, sono stati sviluppati vari strumenti software di sistema, che comprendevano librerie di funzioni comuni, linker, loader, debugger e routine di I/O, utilizzabili come software comune a tutti gli utenti.

Semplici sistemi batch

Poiché le prime macchine erano molto costose, era della massima importanza utilizzarle al meglio: perdere tanto tempo per la schedulazione e la preparazione era inaccettabile.

Per migliorare l'utilizzo delle macchine, si sviluppò il concetto di sistema operativo batch (*a lotti*). Sembra che il primo di tali sistemi (nonché il primo sistema operativo in assoluto) sia stato sviluppato, a metà degli anni '50, dalla General Motors, per un IBM 701[WEIZ81]. L'idea fu in seguito raffinata, e venne implementata su un IBM 704 da un gruppo di clienti della stessa IBM. Dai primi anni '60 diversi produttori svilupparono sistemi operativi batch per i propri computer. IBSYS, il sistema IBM per i computer 7090/7094, merita una menzione particolare per aver direttamente influenzato altri sistemi.

L'idea centrale alla base del semplice schema di elaborazione batch era l'uso di una porzione di software nota come **monitor**. Usando questo tipo di sistema operativo, l'utente non aveva più accesso diretto alla macchina: egli affida il job, su schede o su nastro, a un operatore che ordina i job in lotti sequenzialmente, quindi li pone su un dispositivo di input perché il monitor possa utilizzarli. Ciascun programma è costruito in modo che, alla fine dell'elaborazione, ritorni al monitor, il quale, automaticamente, inizia a caricare il programma successivo.

Per capire come funzioni questo schema, osserviamolo dal punto di vista del monitor e da quello del processore. Nel primo caso, è il monitor stesso a controllare la sequenza di eventi. Perché avvenga questo, la maggior parte del monitor, definita appunto come **monitor residente**, deve risiedere sempre nella memoria principale ed essere utilizzabile per l'esecuzione (Figura 2.3). Il resto del monitor contiene utility e funzioni comuni, caricate come subroutine per il programma utente all'inizio di ogni job che li richieda. Il monitor legge i job uno per volta dal dispositivo di input, ad esempio un lettore di schede o un nastro magnetico; dopo la lettura, il job corrente è posto nell'area del programma utente e gli viene passato il controllo. Quando il job è terminato, il controllo ritorna al monitor, che immediatamente legge il job successivo. Infine, i risultati di ciascun job vengono stampati per essere consegnati all'utente.

Consideriamo ora questa sequenza dal punto di vista del processore. Ad un certo momento, il processore sta eseguendo delle istruzioni dalla porzione della memoria principale che contiene il monitor: tali istruzioni fanno sì che il job successivo sia letto in un'altra porzione della memoria principale. Una volta che un job è stato letto, il processore incontrerà nel monitor un'istruzione di salto, che farà continuare l'esecuzione all'inizio del programma utente, ed eseguirà quindi le istruzioni del programma utente fino a trovare un punto di terminazione o una condizione di errore. In entrambi i casi, il processore preleverà l'istruzione successiva dal programma monitor. Perciò, affermare che "il controllo è passato ad un job" significa semplicemente che il processore sta prelevando ed eseguendo istruzioni dal programma utente, mentre, se sosteniamo che "il controllo è ritornato al monitor", rileviamo come il processore stia prelevando ed eseguendo istruzioni dal programma monitor.

Dovrebbe essere chiaro che il monitor gestisce la schedulazione: un lotto di job viene accodato, e i job sono eseguiti il più rapidamente possibile, senza pause di inattività.

Il monitor gestisce anche il tempo di preparazione del job. Le istruzioni sono contenute, con ciascun job, in una forma primitiva di **job control language** (*linguaggio di controllo dei job*, JCL), un tipo particolare di linguaggio di programmazione, usato per fornire istruzioni al monitor. Un esempio semplice è dato da un programma utente, scritto in FORTRAN, seguito dai dati

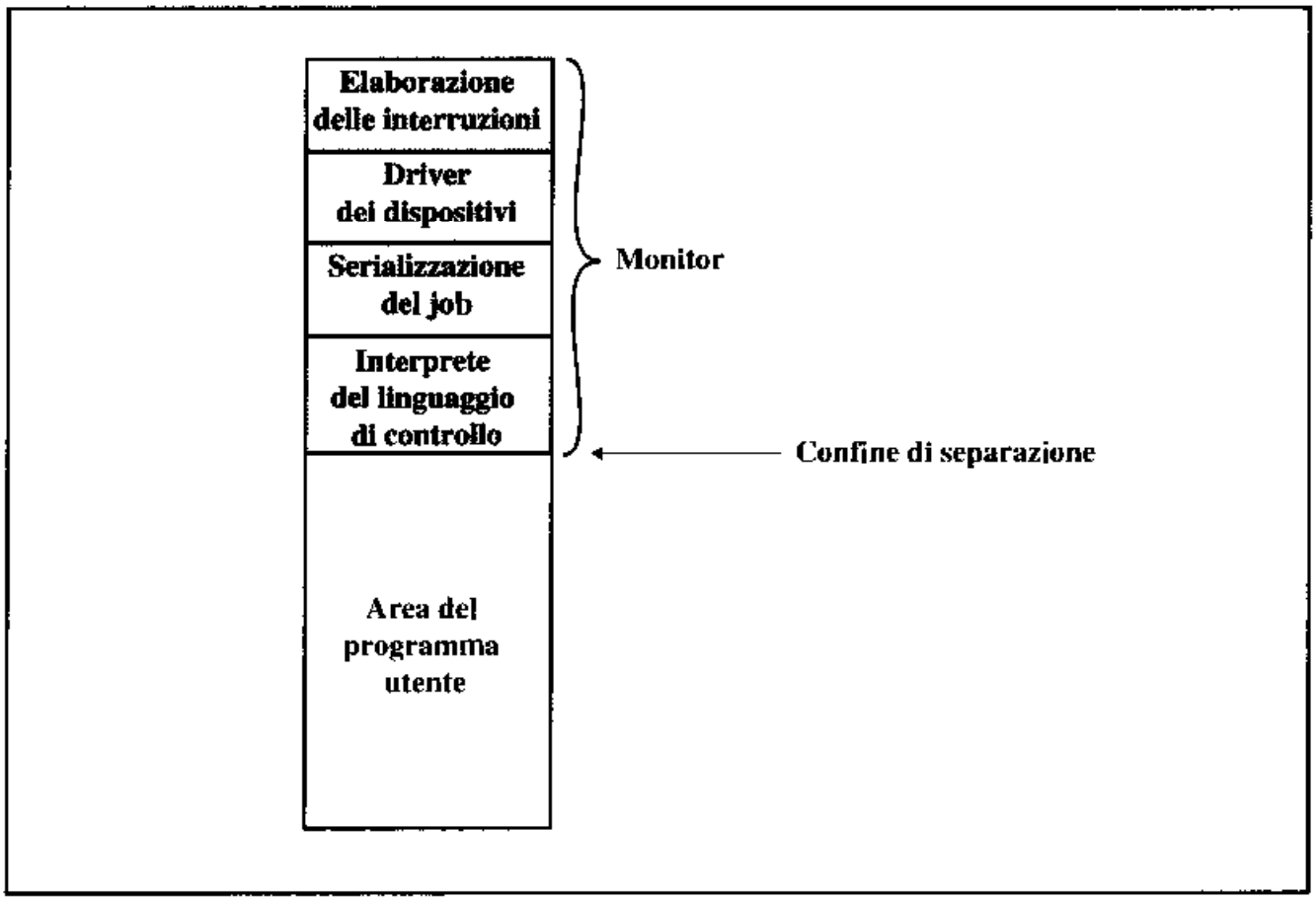


Figura 2.3 *Disposizione in memoria di un monitor residente*

che devono essere usati dal programma stesso; ciascun'istruzione FORTRAN e ciascun dato sta su una scheda separata o su un record del nastro separato. In aggiunta alle linee di codice FORTRAN e a quelle dei dati, il job comprende istruzioni di controllo, contrassegnate dal simbolo iniziale '$'. Il formato complessivo del job si presenta così:

```
$JOB
$FTN
•  ┐
•  │ istruzioni FORTRAN
•  ┘
$LOAD
$RUN
•  ┐
•  │ Dati
•  ┘
$END
```

Per eseguire questo job, il monitor legge la linea $FTN e carica il compilatore appropriato dal dispositivo per la memorizzazione di massa (di solito un nastro); quindi il compilatore traduce il programma utente in codice oggetto, memorizzandolo nella memoria centrale o nella me-

moria di massa. Nel primo caso, l'operazione è definibile come "compila, carica ed esegui" mentre, se il job è memorizzato sul nastro, è richiesta un'operazione di $LOAD. Il monitor legge l'istruzione quando riacquista il controllo dopo l'operazione di compilazione; quindi chiama il loader, che carica il programma oggetto in memoria al posto del compilatore per trasferirgli poi il controllo. In questo modo, un segmento ampio della memoria principale può essere condiviso fra diversi sottosistemi, benché vi risieda, e venga eseguito, un solo sottosistema alla volta.

Durante l'esecuzione del programma utente, ogni istruzione di input determina la lettura di una linea di dati mediante la chiamata di una routine di input del sistema operativo. Tale routine controlla che il programma non legga per errore una linea JCL: se questo accade, si ha un errore, ed il controllo viene trasferito al monitor. Completato con successo o meno il job utente, il monitor riprende a leggere le linee di input fino ad incontrare la successiva istruzione JCL; pertanto, il sistema è protetto dai programmi con troppe o troppo poche linee di dati.

Come si vede, il monitor, o il sistema operativo batch, è un semplice programma che fa affidamento sulla capacità del processore di estrarre istruzioni da varie porzioni della memoria principale, supponendo e rilasciando alternativamente il controllo. Sarebbe opportuno disporre anche di altre caratteristiche hardware:

- **Protezione della memoria**: mentre il programma utente è in esecuzione, non deve alterare l'area di memoria che contiene il monitor. Se questo avvenisse, l'hardware del processore dovrebbe individuare un errore e trasferire il controllo al monitor, che dovrebbe allora fermare il job, stampare un messaggio di errore e caricare il job successivo.
- **Timer**: per impedire ad un singolo job di monopolizzare il sistema si utilizza un timer, avviato all'inizio di ciascun job; quando scade, si verifica un'interruzione e il controllo ritorna al monitor.
- **Istruzioni privilegiate**: certe istruzioni sono definite privilegiate e possono essere eseguite solo dal monitor. Se il processore incontra un'istruzione di questo tipo mentre esegue un programma utente, si verifica un'interruzione di errore. Tra le istruzioni privilegiate ci sono quelle di I/O, in modo che il monitor mantenga il controllo di tutti i dispositivi di I/O, impedendo ad esempio che un programma utente legga per errore le istruzioni di controllo del job successivo. Se un programma utente vuole effettuare un'operazione di I/O, deve richiedere al monitor di farlo in sua vece; se nell'esecuzione di un programma utente s'incontra un'istruzione privilegiata, l'hardware del processore lo considera un errore e trasferisce il controllo al monitor.
- **Interruzioni**: i primi modelli di computer non disponevano di questa risorsa, che fornisce più flessibilità al sistema operativo nel rilasciare e riassegnare il controllo ai programmi utente.

Naturalmente, è possibile progettare un sistema operativo anche su macchine prive di queste caratteristiche, ma i produttori di computer constatarono ben presto che in tal modo i sistemi risultavano difficili da gestire; perciò, anche sistemi operativi batch relativamente primitivi vennero dotati di tali caratteristiche hardware.

Con un sistema operativo batch, il tempo macchina viene distribuito fra l'esecuzione dei programmi utente e quella del monitor, andando perciò incontro a due limitazioni: una parte della memoria principale è ora allocata per il monitor, che utilizza anche parte del tempo macchina. Si tratta di due forme di costi aggiuntivi, che tuttavia non impediscono ad un semplice sistema batch di migliorare l'utilizzo del computer.

Sistemi batch multiprogrammati

Anche con la serializzazione automatica dei job fornita da un semplice sistema batch, il processore rimane spesso inattivo, poiché, in confronto con il processore, i dispositivi di I/O sono lenti. La Figura 2.4 illustra un esempio significativo: il calcolo riguarda un programma che elabora un file di record ed effettua, in media, 100 istruzioni macchina per record. In quest'esempio, il computer passa oltre il 96% del tempo nell'attesa che i dispositivi di I/O completino il trasferimento dei dati! Si veda la Figura 2.5a: in essa, il processore passa una certa quantità di tempo eseguendo lavoro utile, finché raggiunge un'istruzione di I/O e deve quindi attenderne il completamento prima di ripartire.

Una tale inefficienza può essere evitata. Sappiamo già che si deve disporre di memoria sufficiente per contenere il sistema operativo (monitor residente) e un programma utente. Supponiamo ora che ci sia spazio per il sistema operativo e per due programmi utente: quando un job deve aspettare un'operazione di I/O, il processore può passare all'altro job, che molto probabilmente non è in attesa di I/O (Figura 2.5b). Esiste inoltre la possibilità di espandere la memoria per memorizzare tre, quattro o più programmi e passare dall'uno all'altro di essi (Figura 2.5c). Il procedimento è noto come **multiprogrammazione** o **multitasking** e costituisce l'elemento qualificante dei sistemi operativi moderni.

Per renderci conto dei vantaggi della multiprogrammazione, soffermiamoci su un esempio, ricavato da [TURN86]. Si consideri un computer con 256K parole di memoria utilizzabile (non usata dal sistema operativo), un disco, un terminale e una stampante. Tre programmi, JOB1, JOB2 e JOB3, con gli attributi elencati nella Tabella 2.1, sono pronti contemporaneamente per l'esecuzione. Si supponga una richiesta minimale del processore per JOB3 e JOB2, e un uso continuo del disco e della stampante per JOB3. In un semplice ambiente batch, questi job saranno eseguiti in sequenza: JOB1 finisce in 5 minuti, JOB2 aspetta che i 5 minuti siano passati e

Lettura di un record	0.0015 secondi
Esecuzione di 100 istruzioni	0.0001 secondi
Scrittura di un record	0.0015 secondi
TOTALE	0.0031 secondi

$$\text{Percentuale di utilizzo della CPU} = \frac{0.0001}{0.0031} = 0.032 = 3.2\%$$

Figura 2.4 *Esempio di utilizzo di un sistema*

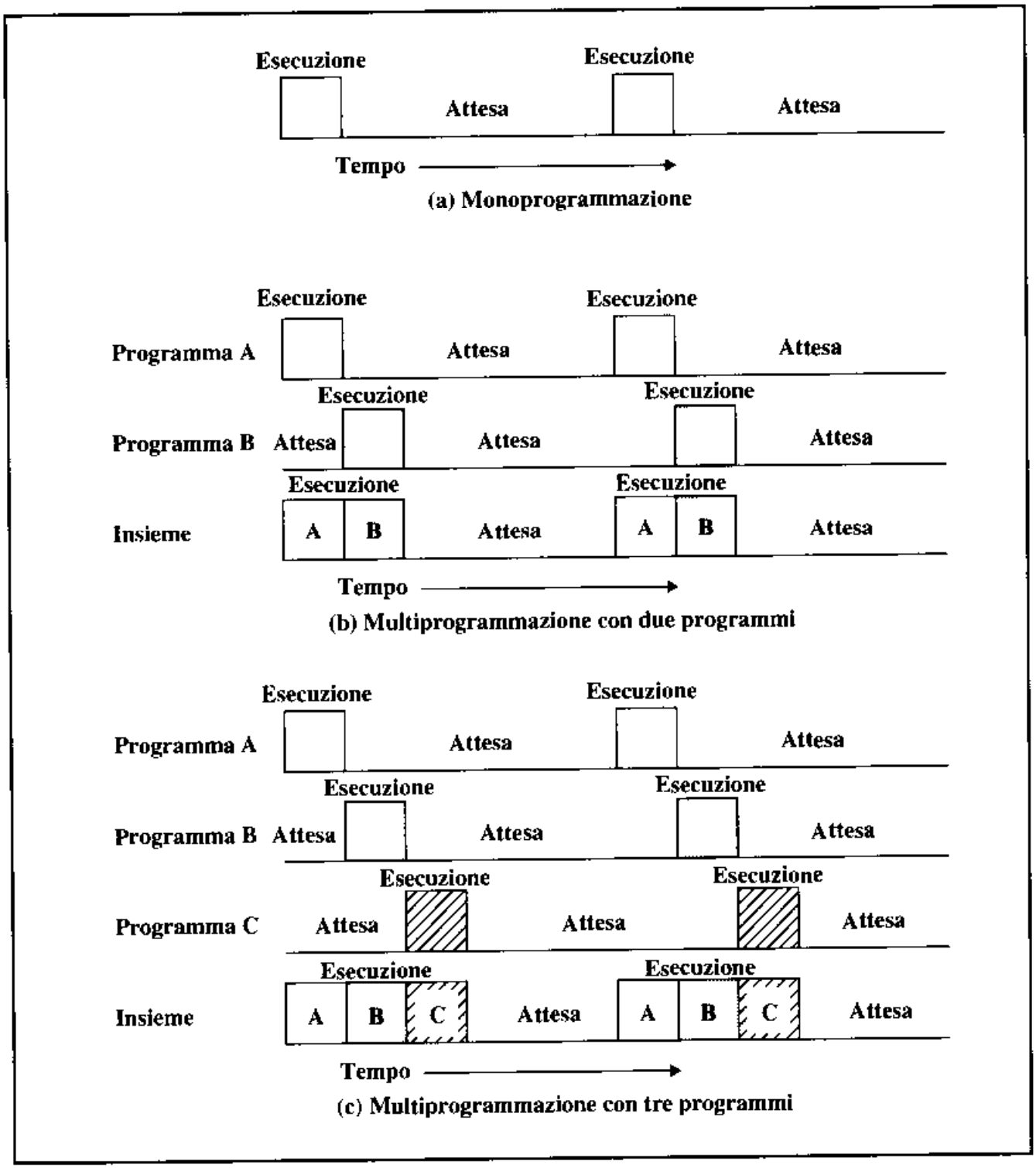


Figura 2.5 *Esempio di multiprogrammazione*

quindi finisce 15 minuti dopo, JOB3 inizia dopo 20 minuti e finisce 30 minuti dopo il momento in cui era pronto per l'esecuzione. L'utilizzazione media delle risorse, il troughput (*numero di job eseguiti nell'unità di tempo*) e i tempi di risposta, compaiono nella colonna "monoprogrammazione" della Tabella 2.2. La Figura 2.6. illustra l'uso dei diversi dispositivi: effettuando le medie sul periodo di tempo richiesto di 30 minuti, appare evidente come tutte le risorse siano largamente sottoutilizzate.

Tabella 2.1 *Esempio di attributi di un'esecuzione di un programma*

	JOB1	JOB2	JOB3
Tipo di job	Calcolo pesante	I/O pesante	I/O pesante
Durata	5 min	15 min	10 min
Memoria richiesta	50K	100K	80K
Ha bisogno del disco?	No	No	Si
Ha bisogno di un terminale?	No	Si	No
Ha bisogno di una stampante?	No	No	Si

Supponiamo ora che i job vengano eseguiti concorrentemente in un sistema operativo multiprogrammato. Poiché fra i job la competizione per accedere alle risorse è modesta, tutti e tre possono essere eseguiti in un tempo vicino al minimo, mentre coesistono con gli altri nel computer (supponendo di assegnare a JOB2 e a JOB3 tempo di processore sufficiente a mantenere attive le proprie operazioni di input e output). JOB1 richiederà ancora 5 minuti per finire, entro i quali però JOB2 avrà completato un terzo del suo lavoro, e JOB3 sarà arrivato a metà: tutti e tre i job finiscono entro 15 minuti. La colonna "multiprogrammazione" (Tabella 2.2), ottenuta dall'istogramma contenuto nella Figura 2.7, evidenzia il miglioramento.

Al pari di un sistema batch semplice, un sistema batch multiprogrammato deve sfruttare determinate caratteristiche hardware del computer; quella più utile per la multiprogrammazione è un hardware che supporti gli interrupt di I/O e il DMA. Con tale hardware, il processore può effettuare un comando di I/O per un job e proseguire con l'esecuzione di un altro job mentre l'operazione di I/O è presa in carico dal controller del dispositivo. Completata l'operazione di I/O, il processore viene interrotto e il controllo passa a un programma del sistema operativo per la gestione dell'interruzione. Il sistema operativo passerà poi il controllo ad un altro job.

Se confrontati con i sistemi operativi a singolo programma o **monoprogrammati**, quelli multiprogrammati sono piuttosto complessi. Per avere diversi job pronti per l'esecuzione, è necessario mantenerli nella memoria principale, utilizzando qualche forma di **gestione della**

Tabella 2.2 *Effetti della multiprogrammazione sull'utilizzo delle risorse*

	Monoprogrammazione	Multiprogrammazione
Uso del processore	17%	33%
Uso della memoria	30%	67%
Uso del disco	33%	67%
Uso della stampante	33%	67%
Tempo trascorso	30 min	15 min
Tasso di throughput	6 job/ora	12 job/ora
Tempo medio di risposta	18 min	10 min

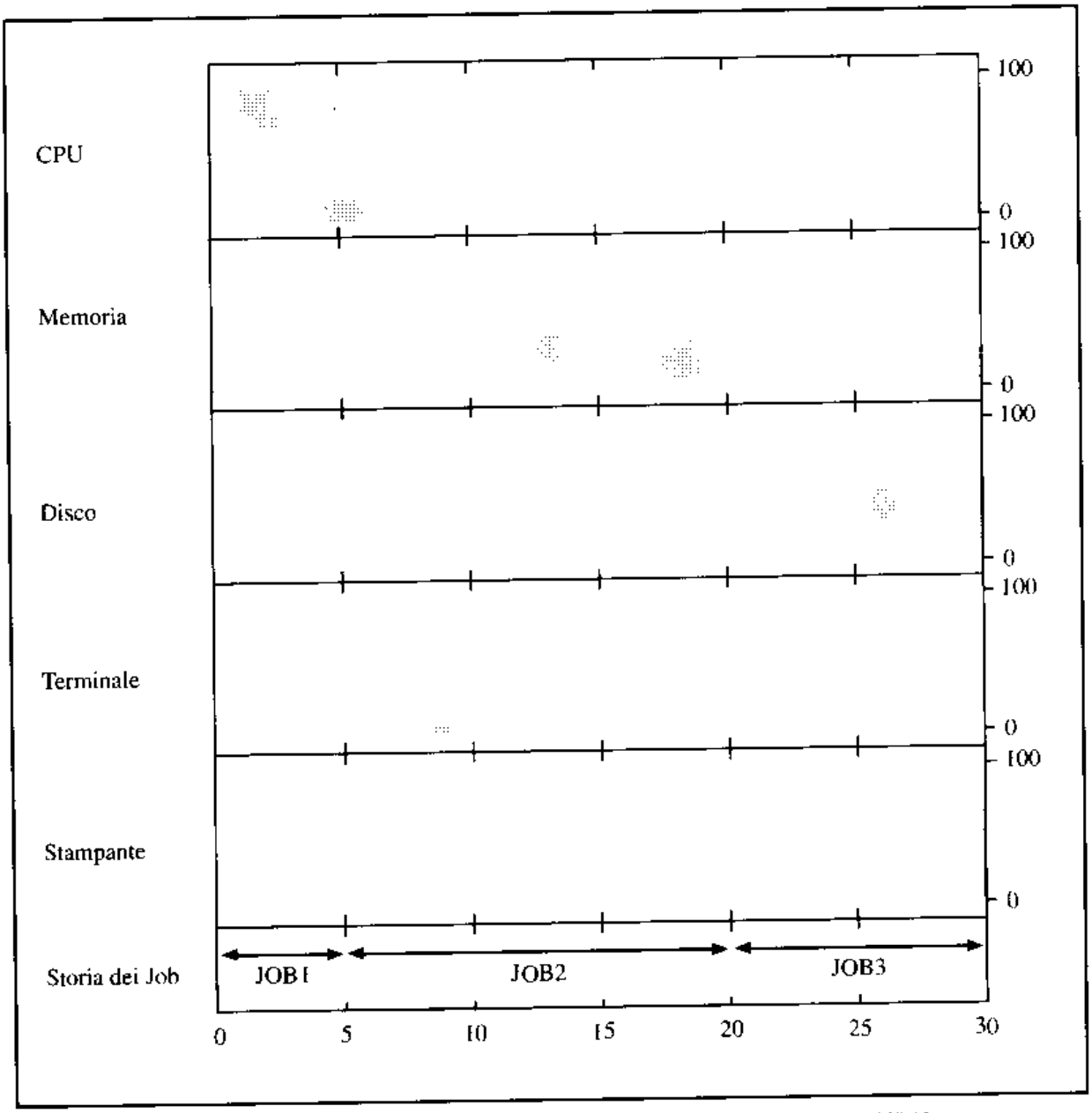


Figura 2.6 *Istogramma di utilizzazione monoprogrammata delle risorse [TURN86]*

memoria; inoltre, fra diversi job pronti per l'esecuzione, il processore deve decidere quale mandare in esecuzione, il che richiede un algoritmo di schedulazione. Tali aspetti della questione verranno esaminati nel corso di questo capitolo.

Sistemi time-sharing

Con l'uso della multiprogrammazione, l'elaborazione batch può essere abbastanza efficiente, anche se, per molti lavori, resta preferibile consentire all'utente di interagire direttamente con il computer. In certi casi, infatti, come ad esempio l'elaborazione delle transazioni, la modalità interattiva appare essenziale.

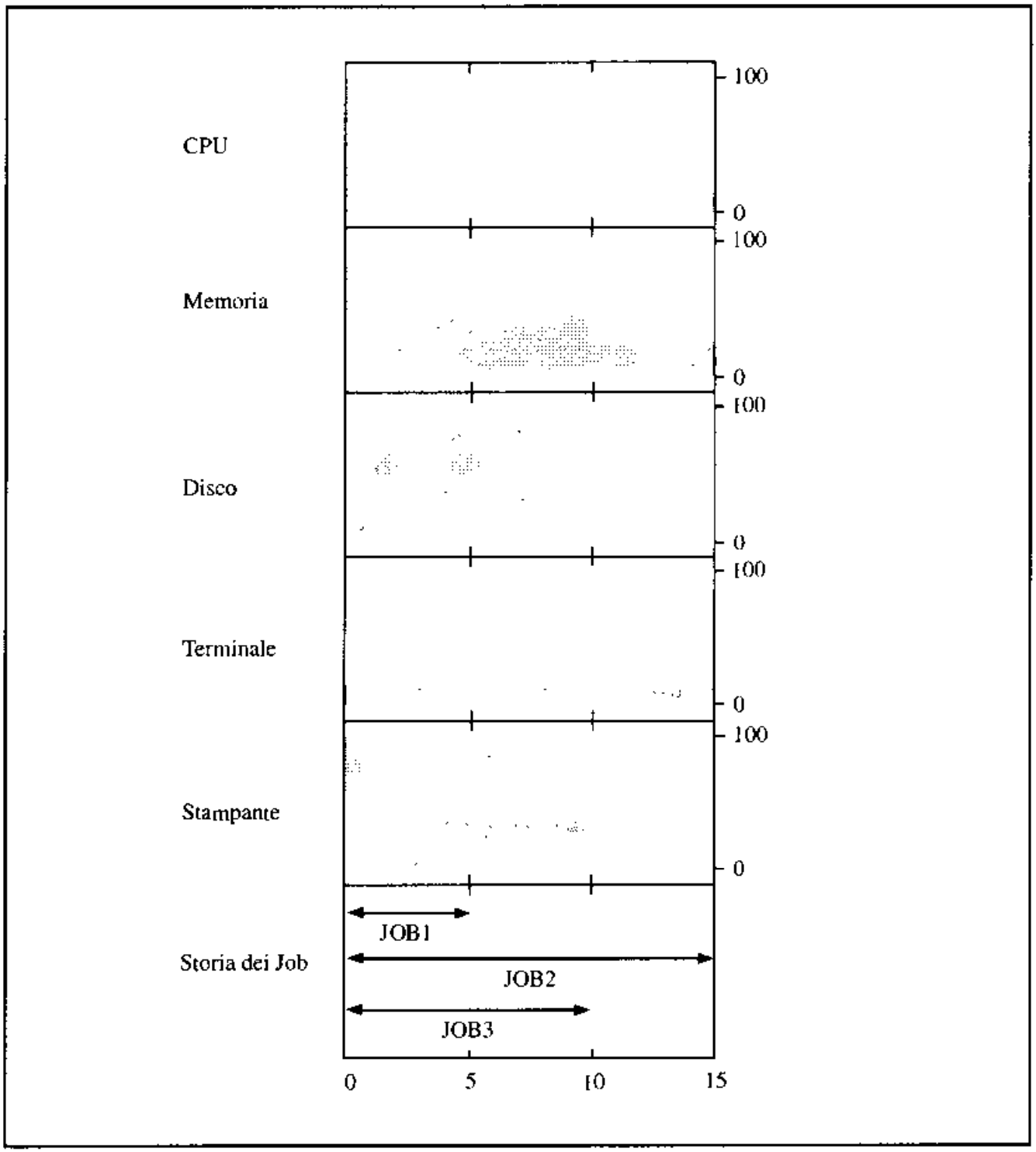


Figura 2.7 *Istogramma di utilizzazione multiprogrammata delle risorse [TURN86]*

Oggi, un microcomputer dedicato è spesso in grado di soddisfare le esigenze dell'elaborazione interattiva, come invece non era possibile negli anni '60, quando la maggioranza dei computer era grande e costosa: in alternativa, è stato sviluppato il time-sharing (*sistema a condivisione di tempo*).

La multiprogrammazione infatti permette al processore di gestire più job batch alla volta, e la si può dunque utilizzare per gestire molti job interattivi: questa tecnica viene definita time-

sharing, poiché il tempo di processore è condiviso fra utenti multipli. In un sistema time-sharing, più utenti hanno accesso simultaneo attraverso dei terminali, mentre il sistema operativo alterna l'esecuzione di ciascun programma utente per un periodo breve, detto quanto, di elaborazione. Quindi, se n utenti richiedono attivamente un servizio nel medesimo tempo, ciascuno di essi disporrà in media di $1/n$ della velocità effettiva del computer, senza contare l'overhead dovuto al sistema operativo. Comunque, rispetto al tempo di reazione umano relativamente lungo, il tempo di risposta, su un sistema progettato in modo opportuno, dovrebbe essere confrontabile a quello di un computer dedicato.

Il batch multiprogrammato e il time-sharing utilizzano entrambi la multiprogrammazione: nella Tabella 2.3 sono elencate le differenze più rilevanti.

Uno dei primi sistemi operativi time-sharing è stato il Compatible Time-Sharing System (CTSS) [CORB62], sviluppato al MIT da un gruppo noto come Project MAC (Machine Aided Cognition o Multiple Access Computers), inizialmente (1961) per l'IBM 709 e quindi trasferito su di un IBM 7094.

Se confrontato con i sistemi successivi, CTSS era piuttosto primitivo. Il sistema era eseguito su una macchina con 32K parole da 36 bit in memoria principale, di cui il monitor residente occupava 5K. Dovendo assegnare il controllo ad un utente interattivo, il programma utente e i dati venivano caricati nei restanti 27K di memoria principale: un clock di sistema generava interruzioni ad una frequenza di circa uno ogni 0.2 secondi, e ad ogni interruzione di clock il sistema operativo riacquistava il controllo e poteva assegnare il processore ad un diverso utente. Quindi, a intervalli di tempo regolari, l'utente corrente veniva sospeso e messo in attesa, e un altro utente veniva caricato in memoria. Al fine di preservare lo stato del vecchio utente, per poterne poi riprendere l'esecuzione, i suoi programmi e dati venivano scritti sul disco, prima che programmi e dati del nuovo utente fossero letti e caricati in memoria; lo spazio di memoria del vecchio utente veniva poi recuperato al successivo turno di esecuzione.

Per minimizzare il traffico da e per il disco, la memoria utente veniva scritta su disco solo quando il programma entrante doveva sovrascriverla. Questo principio è illustrato nella Figura 2.8. Supponiamo di avere quattro utenti interattivi con le seguenti necessità di memoria:

- JOB1: 15K
- JOB2: 20K
- JOB3: 5K
- JOB4: 10K

Tabella 2.3 *Confronto fra la multiprogrammazione batch e time-sharing*

	Multiprogrammazione batch	**Time sharing**
Obiettivo principale	Massimizzare l'uso del processore	Minimizzare il tempo di risposta
Sorgente delle istruzioni del sistema operativo	Linguaggio di controllo dei job: istruzioni fornite con il job stesso	Comandi inseriti al terminale

Inizialmente il monitor carica JOB1 e gli trasferisce il controllo (Figura 2.8a); in seguito, il monitor decide di trasferire il controllo a JOB2, che richiede più memoria di JOB1. Perciò, bisogna prima scrivere JOB1 sul disco, e quindi caricare JOB2 (Figura 2.8b); successivamente JOB3 viene caricato per l'esecuzione. Comunque, poiché JOB3 è più piccolo di JOB2, una porzione di JOB2 può rimanere in memoria, riducendo il tempo di scrittura su disco (Figura 2.8c). Più tardi, il monitor decide nuovamente di trasferire il controllo a JOB1. Quando JOB1 viene nuovamente caricato in memoria (Figura 2.8d), occorre salvare un'ulteriore porzione di JOB2; e quando viene caricato JOB4, una parte di JOB1 e la porzione di JOB2 che rimane in memoria sono mantenute (Figura 2.8e). A questo punto, se JOB1 o JOB2 vengono attivati, sarà necessario solo un caricamento parziale. In quest'esempio è JOB2 che viene eseguito subito dopo, richiedendo che JOB4 e la porzione rimanente di JOB1 siano scritte sul disco e che la porzione mancante di JOB2 venga letta e caricata in memoria.

L'approccio CTSS, primitivo se confrontato con gli attuali sistemi time-sharing, risultava però funzionale ed era estremamente semplice, il che minimizzava la dimensione del monitor. Poiché un job era sempre caricato nella medesima locazione in memoria, non si doveva ricorrere, all'atto del caricamento, a tecniche di rilocazione (di cui ci occuperemo più avanti): scrivendo solo il necessario, si minimizzava l'attività del disco. Eseguito su un 7094, CTSS supportava un massimo di 32 utenti.

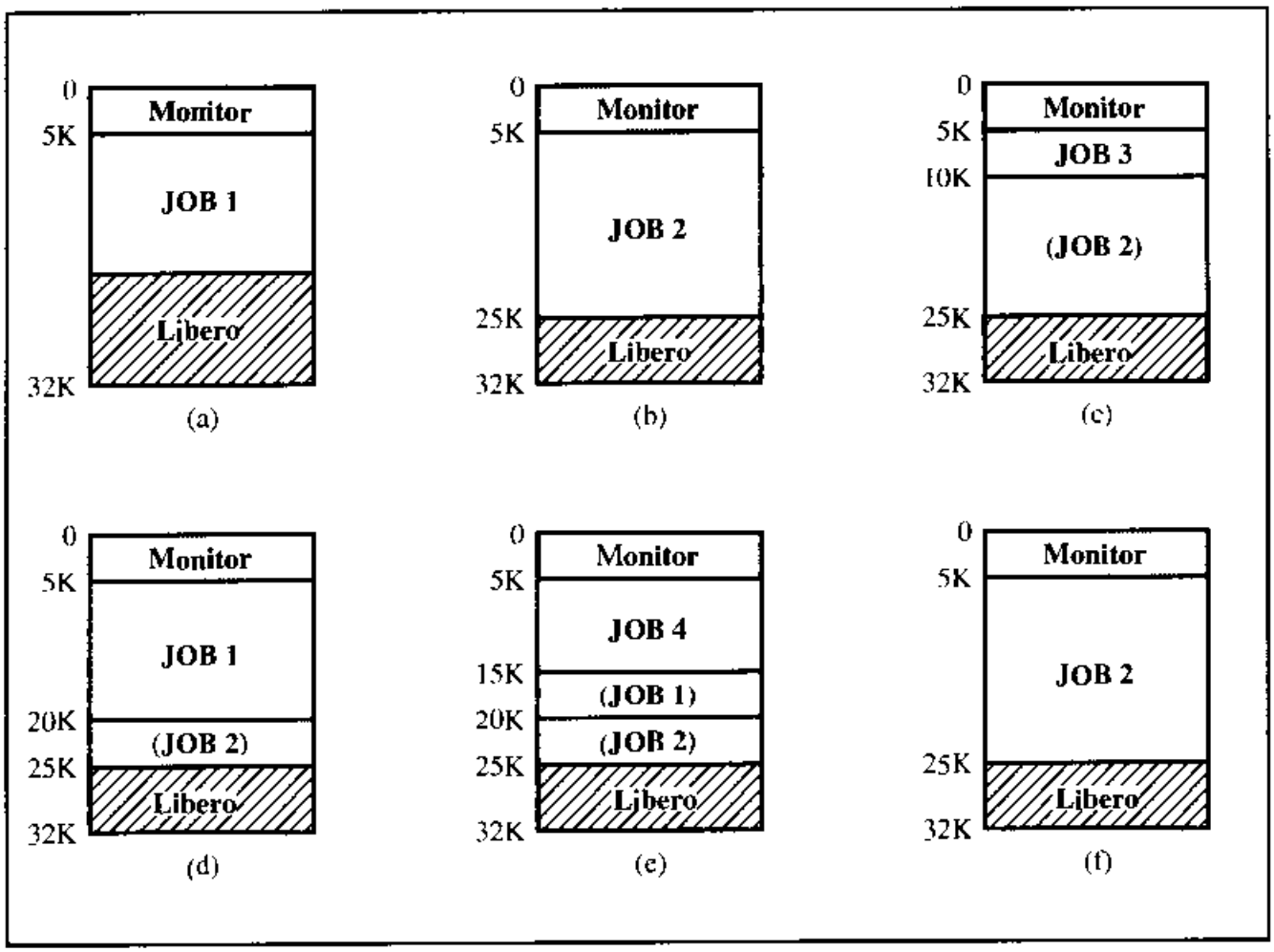


Figura 2.8 *Funzionamento del sistema CTSS*

Il time-sharing e la multiprogrammazione sollevano una quantità di nuovi problemi relativi al sistema operativo. Ad esempio, se più job risiedono in memoria, occorre impedire che interferiscano l'uno con l'altro, modificando reciprocamente i propri dati; nel caso di più utenti interattivi, il file system dev'essere protetto, in modo che solo gli utenti autorizzati abbiano accesso ad un dato file, e bisogna inoltre gestire la competizione per l'accesso a risorse, quali stampanti e dispositivi di memorizzazione di massa. Nel corso di questo libro, si parlerà anche di questi nuovi problemi e delle loro possibili soluzioni.

2.3 Aspetti principali

I sistemi operativi sono fra i tipi di software più complessi, poiché fanno propria la sfida di conciliare obiettivi, difficili da raggiungere e in certi casi in conflitto l'uno con l'altro: la convenienza, l'efficienza e la capacità di evoluzione. [DENN80a] elenca cinque aspetti fondamentali nello sviluppo dei sistemi operativi:

- Processi
- Gestione della memoria
- Protezione e sicurezza dell'informazione
- Schedulazione e gestione delle risorse
- Struttura del sistema.

Ciascuno di essi si fonda su principi o astrazioni, sviluppati per affrontare problemi complessi. Prese insieme, queste cinque aree coprono la totalità della progettazione e dell'implementazione dei moderni sistemi operativi. In questa sezione se ne fornisce un breve riassunto, che vale come sguardo d'assieme sui prossimi capitoli di questo testo.

Processi

Nella struttura dei sistemi operativi, quello di processo è un concetto fondamentale. Questo termine, più generale di job, è stato usato la prima volta dai progettisti di Multics negli anni '60, e se ne conoscono molte definizioni, tra cui:

- Un programma in esecuzione
- L' "anima" di un programma
- L'entità assegnata ad un processore e da esso eseguita.

Tale concetto dovrebbe comunque farsi più chiaro nel corso della trattazione.

Tre principali linee di sviluppo dei sistemi di elaborazione hanno creato problemi nella temporizzazione e sincronizzazione, contribuendo allo sviluppo del concetto di processo: le operazioni del batch multiprogrammato, il time-sharing e i sistemi di transazione in tempo reale.

Come abbiamo visto, la multiprogrammazione è stata progettata per mantenere processore e dispositivi di I/O, compresi quelli di memorizzazione di massa, simultaneamente occupati, al fine di conseguire la massima efficienza. Il meccanismo chiave è questo: in risposta a segnali che indicano il completamento di operazioni di I/O, il processore passa alternativamente fra i vari processi che risiedono nella memoria principale.

Una seconda linea di sviluppo è stato il time-sharing general-purpose, il cui obiettivo principale è la velocità di risposta alle necessità di ogni utente: più utenti utilizzano il sistema contemporaneamente, riducendo così i costi. Si tratta di obiettivi compatibili, a causa del tempo di reazione dell'utente relativamente lungo. Ad esempio, se un utente necessita in media di due secondi di tempo di elaborazione per minuto, all'incirca una trentina di utenti potrebbero condividere il medesimo sistema senza interferenze rilevanti, tenendo ovviamente conto dell'overhead dovuto al sistema operativo.

Anche i sistemi di elaborazione delle transazioni in tempo reale hanno avuto grande importanza. In questo caso, un certo numero di utenti esegue interrogazioni o aggiornamenti di un data base, come ad esempio, il sistema di prenotazione di una linea aerea. La differenza sostanziale fra i sistemi di questo tipo e quelli time-sharing sta nel fatto che i primi sono limitati ad una o poche applicazioni, mentre gli utenti di un sistema time-sharing possono dedicarsi allo sviluppo di programmi, all'esecuzione di job e all'uso di diverse applicazioni. In entrambi i casi, l'aspetto prevalente è il tempo di risposta.

Lo strumento principale a disposizione dei programmatori per sviluppare i primi sistemi interattivi multiprogrammati e multiutente, è stato l'interrupt. Al verificarsi di un determinato evento, ad esempio il completamento di un'operazione di I/O, l'attività di un job poteva essere sospesa; il processore salvava il contesto (cioè il contatore di programma e gli altri registri) e saltava ad una routine per la gestione dell'interruzione, che determinava la natura dell'interruzione, la elaborava e ripristinava quindi l'elaborazione dei programmi utente con il job interrotto o qualche altro job.

La progettazione del software di sistema per coordinare tali attività diverse appariva veramente complessa. Con molti job attivi in ogni momento, ciascuno dei quali comportava numerosi passi da eseguirsi in sequenza, diveniva impossibile analizzare tutte le possibili combinazioni di sequenze di eventi. In assenza di principi sistematici di coordinamento e cooperazione fra le attività, i programmatori, fondandosi sulla conoscenza dell'ambiente che il sistema operativo doveva controllare, sono ricorsi a metodi ad hoc, andando tuttavia incontro ad errori di programmazione subdoli, i cui effetti erano osservabili solo al verificarsi di determinate sequenze di azioni relativamente rare. Questi errori, inoltre, erano difficili da diagnosticare, poiché bisognava distinguerli da quelli delle applicazioni e da quelli hardware; trovato l'errore, era tuttavia problematico individuarne la causa, poiché non era facile riprodurre con precisione le condizioni in cui si era manifestato. In sintesi, questi errori sono provocati da quattro cause principali [DENN80a]:

- **Sincronizzazione impropria**: accade spesso che una routine debba essere sospesa in attesa di un evento nel sistema. Ad esempio, un programma inizia una lettura di I/O e, prima di procedere, deve aspettare che i dati siano disponibili in un buffer: in tal caso, è richiesto un segnale da un'altra routine. La progettazione impropria del meccanismo di gestione dei segnali può portare alla perdita di segnali o alla duplicazione di quelli ricevuti.

- **Fallimento della mutua esclusione**: spesso più di un utente o programma tentano di usare contemporaneamente una risorsa condivisa. Ad esempio, in un sistema di prenotazioni di linee aeree due utenti leggono contemporaneamente il database, trovando un posto disponibile, e cercano entrambi di aggiornare il database per prenotarlo. Se questi accessi non sono controllati, può verificarsi un errore. Deve esistere un qualche meccanismo di mutua esclusione, che permetta solo ad una routine alla volta di effettuare una transazione su una parte dei dati. È difficile verificare la correttezza dell'implementazione della mutua esclusione, considerando tutte le possibili sequenze di eventi.
- **Operazioni del programma non determinate**: in un sistema condiviso, i risultati di un particolare programma di norma dovrebbero dipendere solo dall'input del programma stesso e non dall'attività degli altri programmi. Quando i programmi condividono la memoria e la loro esecuzione è interallacciata, possono interferire l'un l'altro, sovrascrivendo aree comuni di memoria in modi imprevedibili; pertanto, l'ordine in cui vari programmi sono schedulati può influire sull'esito di uno qualsiasi di essi.
- **Stallo (*deadlock*)**: è possibile che due o più programmi siano sospesi in reciproca attesa: supponiamo ad esempio che entrambi richiedano due dispositivi di I/O per effettuare un'operazione, come una copia da disco a nastro. Ciascuno dei due programmi acquisisce il controllo di uno dei due dispositivi, e aspetta che l'altro rilasci la risorsa desiderata: tale condizione di stallo può dipendere dalle diverse possibilità di temporizzazione nell'allocazione e rilascio delle risorse.

Per affrontare questi problemi, occorre monitorare e controllare in modo sistematico i vari programmi eseguiti dal processore: il concetto di processo fornisce la base a questo controllo. Si può pensare che un processo sia costituito da tre componenti:

- Un programma eseguibile.
- I dati associati che servono al programma (variabili, spazio di lavoro, buffer ecc.).
- Il contesto di esecuzione del programma.

Quest'ultimo elemento è essenziale: il contesto di esecuzione comprende tutte le informazioni necessarie al sistema operativo per gestire il processo, e ciò di cui necessita il processore per eseguirlo correttamente. Quindi, il contesto comprende i contenuti dei vari registri del processore, come il contatore di programma e i registri dei dati, nonché l'informazione utilizzata dal sistema operativo, come la priorità del processo, o la condizione di attesa del completamento di un particolare evento di I/O.

La Figura 2.9 illustra come possono essere implementati i processi. Due processi, A e B, risiedono in diverse parti della memoria principale: per entrambi, viene allocato un blocco di memoria che contiene il programma, i dati e le informazioni di contesto, e ciascuno è registrato in una lista dei processi, costruita e gestita dal sistema operativo. Questa lista contiene una entry per ciascun processo, che comprende un puntatore alla locazione del blocco di memoria che lo contiene. L'entry può anche includere, completamente o in parte, il contesto di esecuzione. Il resto del contesto di esecuzione è memorizzato con il processo stesso. Il registro indice del

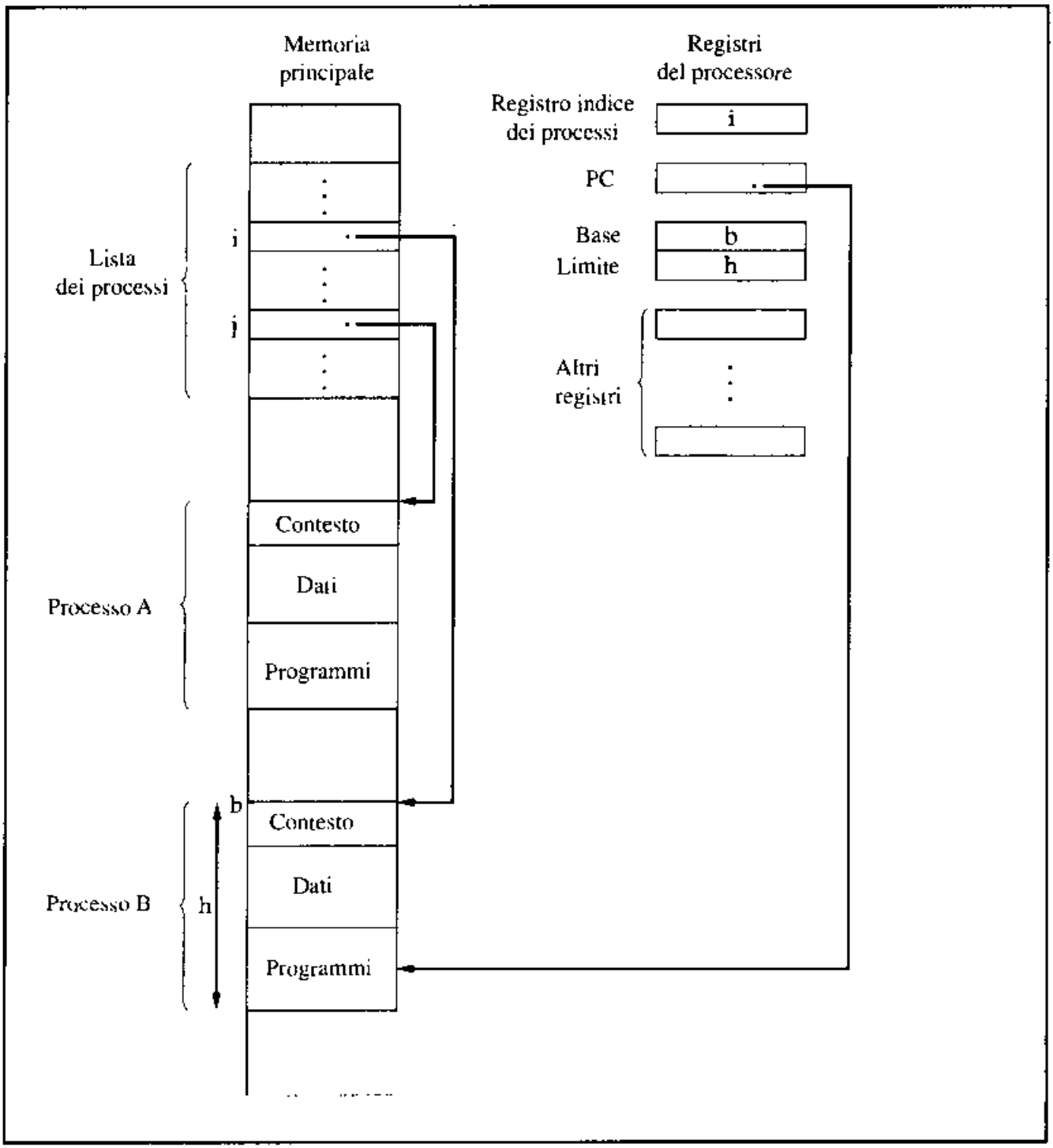


Figura 2.9 *Implementazione tipica di un processo*

processo contiene l'indice, nella lista dei processi, di quel processo che controlla attualmente il processore. Il program counter punta all'istruzione successiva del processo da eseguire, e i registri base e limite definiscono la regione di memoria occupata dal processo. Il program counter e tutti i riferimenti ai dati sono interpretati relativamente al registro base, e non devono eccedere il valore del registro limite, prevenendo così interferenze fra i processi.

Nella figura, il registro indice del processo indica che B è in esecuzione; A, che era prima in esecuzione, è stato temporaneamente interrotto. Il contenuto di tutti i registri al momento del-

l'interruzione di A è stato salvato nel suo contesto di esecuzione; più tardi il processore può passare ad un altro processo e ripristinarne l'esecuzione. In tale passaggio (*switch*) si memorizza il contesto di B e si ripristina quello di A; quando il program counter è caricato con un valore che punta all'area di programma di A, esso riprenderà automaticamente l'esecuzione.

Quindi, il processo è realizzato come una strottura dati. Un processo può essere in esecuzione o in attesa, e il suo intero "stato" è contenuto nel suo contesto. Questa struttura permette lo sviluppo di tecniche potenti per assicurare il coordinamento e la cooperazione fra i processi; è possibile progettare e incorporare nel sistema operativo nuove caratteristiche (ad esempio, la priorità) espandendo il contesto per comprendere eventuali nuove informazioni, necessarie per supportare tali caratteristiche. Nel corso della lettura, incontreremo diversi esempi in cui questa struttura del processo è impiegata per risolvere i problemi generati dalla multiprogrammazione e dalla condivisione delle risorse.

Gestione della memoria

Gli utenti hanno bisogno di un ambiente di elaborazione che supporti la programmazione modulare e l'utilizzo flessibile dei dati, mentre gli amministratori di sistema necessitano di controllare l'allocazione della memoria in modo efficiente ed ordinato. Per soddisfare queste esigenze, il sistema operativo deve assolvere cinque compiti principali, inerenti la gestione della memorizzazione:

- **Isolamento dei processi**: il sistema operativo deve impedire a processi indipendenti di interferire tra di loro, relativamente ai dati e alla memoria.
- **Allocazione e gestione automatica della memoria**: in base alle esigenze, i programmi dovrebbero essere allocati dinamicamente attraverso la gerarchia di memoria, e tale allocazione dovrebbe essere trasparente (cioè invisibile) al programmatore, che risulta così sollevato dalle incombenze relative alle limitazioni della memoria; il sistema operativo guadagna in efficienza assegnando la memoria ai job solo quando necessario.
- **Supporto per la programmazione modulare**: i programmatori dovrebbero poter definire i moduli di programma e crearne, distruggerne e alterarne dinamicamente la dimensione.
- **Protezione e controllo dell'accesso**: la condivisione della memoria, ad ogni livello della gerarchia, consente ad un programma di indirizzare lo spazio di memoria di un altro. In certi casi, quando la condivisione è necessaria per una particolare applicazione, ciò è positivo. ma in altri casi mette in pericolo l'integrità dei programmi e quella del sistema operativo stesso, che deve permettere a utenti diversi l'accesso a porzioni di memoria secondo diverse modalità.
- **Memorizzazione a lungo termine**: molti utenti e molte applicazioni richiedono stromenti per memorizzare l'informazione per lunghi periodi di tempo.

Tipicamente, i sistemi operativi soddisfano queste necessità con la memoria virtuale e i servizi del file system. La memoria virtuale è una funzionalità che permette ai programmi di indirizzare la memoria da un punto di vista logico, senza preoccuparsi della quantità di memoria

fisicamente disponibile. Quando un programma è in esecuzione, solamente una parte del programma e dei dati può effettivamente risiedere nella memoria principale; altre porzioni del programma e dei dati sono mantenute nei blocchi sul disco. Vedremo nei prossimi capitoli che questa separazione della memoria indirizzabile, secondo un punto di vista fisico e logico, fornisce al sistema operativo un mezzo potente per conseguire i suoi obiettivi.

Il file system implementa una memorizzazione a lungo termine, con l'informazione memorizzata in oggetti etichettati con nome, chiamati file. Quello di file è un concetto conveniente per il programmatore, e costituisce un'utile unità di controllo dell'accesso e di protezione per il sistema operativo.

La Figura 2.10 presenta una descrizione generale di un sistema di memorizzazione gestito da un sistema operativo. L'hardware del processore, insieme con il sistema operativo, fornisce all'utente una sorta di "processore virtuale", che ha accesso alla memoria virtuale. Quest'ultima può essere uno spazio di indirizzamento lineare, o una collezione di segmenti, che sono blocchi di lunghezza variabile di indirizzi contigui. In entrambi i casi, le istrozioni del linguaggio di programmazione possono riferirsi a locazioni del programma e dei dati nella memoria virtuale. L'isolamento dei processi può essere ottenuto affidando a ciascuno di essi blocchi differenti di memoria virtuale, tali da non sovrapporsi tra di loro, mentre la condivisione tra processi si può ottenere sovrapponendo porzioni di due spazi di memoria virtuale. I file sono mantenuti in un supporto di memorizzazione a lungo termine. File e loro parti si possono copiare nella memoria virtuale per essere poi elaborati dai programmi.

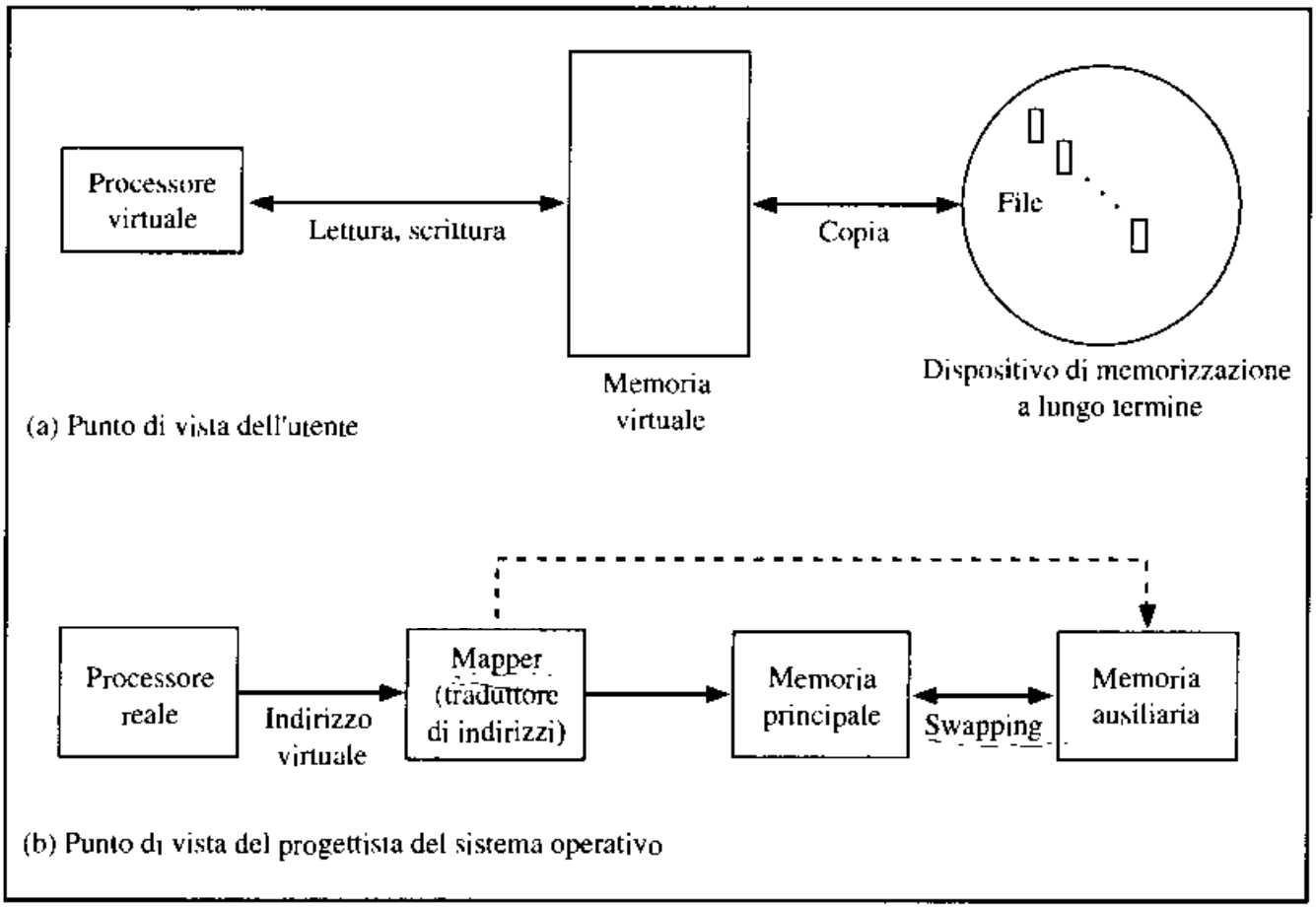


Figura 2.10 *Due punti di vista per i sistemi di memorizzazione [DENN80a]*

La Figura 2.10 illustra anche il punto di vista del progettista relativo alla memorizzazione. Il sistema di memorizzazione comprende la memoria principale, direttamente indirizzabile (dalle istruzioni macchina), e la memoria ausiliaria più lenta, cui si accede indirettamente attraverso blocchi caricati nella memoria principale. Fra il processore e la memoria è interposto un mapper (*hardware per la traduzione degli indirizzi*); i programmi fanno riferimento alle locazioni tramite gli indirizzi virtuali, che sono mappati in indirizzi reali della memoria principale. Nel caso di un riferimento a un indirizzo virtuale che non si trova nella memoria reale, allora una parte dei contenuti della memoria reale viene copiata nella memoria ausiliaria, e il blocco di dati desiderato viene caricato nella memoria principale. Durante quest'attività, il processo che ha generato il riferimento alla memoria dev'essere sospeso. È compito del progettista sviluppare un meccanismo di traduzione degli indirizzi che generi solo un piccolo overhead, e una politica di allocazione della memoria che minimizzi il traffico fra i livelli di memoria stessi.

Protezione dell'informazione e sicurezza

L'uso crescente dei sistemi time-sharing e, negli ultimi anni, delle reti di computer, ha ingigantito i problemi relativi alla protezione dell'informazione. La natura del rischio cui è sottoposta una certa istituzione varia grandemente a seconda delle circostanze. Comunque, esiste una serie di strumenti per computer e per sistemi operativi che supportano diversi meccanismi di protezione e sicurezza. In generale, i problemi riguardano l'accesso ai sistemi di elaborazione e all'informazione memorizzata in essi.

In relazione alla sicurezza ed alla protezione, le principali operazioni di competenza dei sistemi operativi ricadono in queste tre categorie:

- **Controllo dell'accesso**: regola l'accesso degli utenti al sistema, ai sottosistemi e ai dati, e l'accesso dei processi alle diverse risorse e oggetti del sistema.
- **Controllo del flusso dell'informazione**: regola il flusso dei dati nel sistema e verso gli utenti.
- **Certificazione**: verifica che i meccanismi di controllo dell'accesso e del flusso vengano eseguiti in accordo alle loro specifiche, ed applichino le politiche di sicurezza e protezione prescelte.

Schedulazione e gestione delle risorse

Un compito essenziale dei sistemi operativi è la gestione delle diverse risorse disponibili (spazio della memoria principale, dispositivi di I/O, processori) e lo schedulare il loro utilizzo da parte dei diversi processi attivi. Una politica di allocazione delle risorse e di schedulazione deve considerare tre fattori:

- **Equità**: tutti i processi che sono in competizione per l'utilizzo di una particolare risorsa dovrebbero godere della medesima possibilità di accesso a quella risorsa; ciò è particolarmente vero per job della stessa classe, ovvero job con simili richieste, dal costo analogo.

- **Tempo di risposta differenziale**: è possibile, d'altra parte, che il sistema operativo debba discriminare fra classi differenti di job, che necessitano di servizi diversi, prendendo decisioni di allocazione e schedulazione nel rispetto dell'insieme complessivo delle richieste ed in modo dinamico. Ad esempio, se un processo è in attesa di usare un dispositivo di I/O, il sistema operativo può schedulare quel processo perché sia eseguito appena possibile, liberando così il dispositivo per ulteriori richieste da parte di altri processi.
- **Efficienza**: entro i vincoli dell'equità e dell'efficienza, il sistema operativo dovrebbe cercare di massimizzare il throughput, minimizzare il tempo di risposta, e, in caso di time-sharing, di soddisfare il maggior numero possibile di utenti.

La schedulazione e la gestione delle risorse sono essenzialmente problemi di ricerca operativa e se ne possono applicare i risultati matematici. Inoltre, è importante misurare l'attività del sistema, al fine di monitorare le prestazioni e di effettuare eventuali correzioni.

La Figura 2.11 presenta i principali elementi di un sistema operativo coinvolti nella schedulazione dei processi e nell'allocazione delle risorse in un ambiente multiprogrammato. Il sistema operativo gestisce un insieme di code, ciascuna delle quali è semplicemente una lista di processi in attesa di una risorsa. La coda a breve termine si compone di processi che risiedono

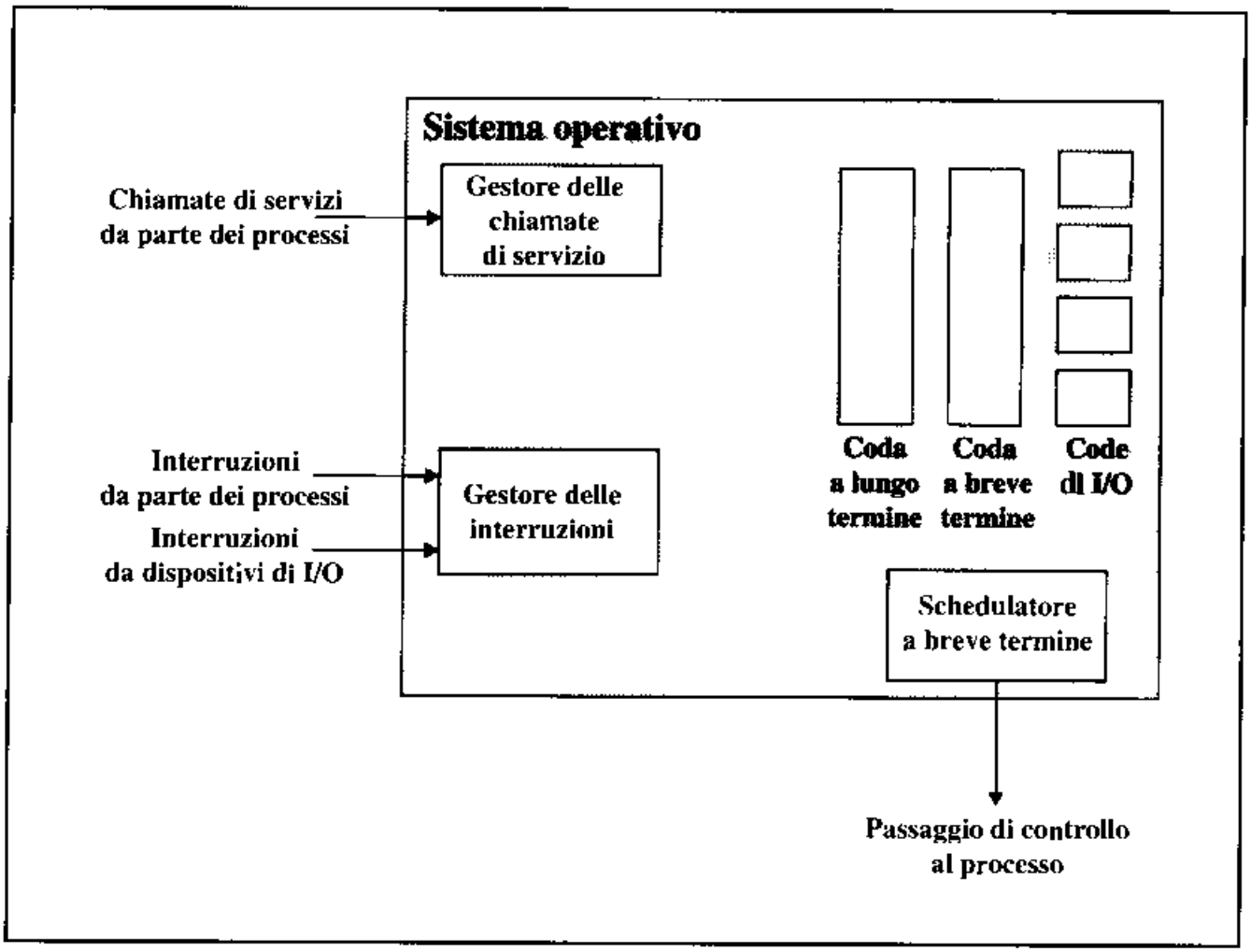


Figura 2.11 *Elementi chiave di un sistema operativo per la multiprogrammazione*

nella memoria principale (almeno per la parte minima essenziale) e che sono pronti per l'esecuzione. Ognuno di questi processi potrebbe usare il processore immediatamente. È compito dello schedulatore a breve termine, o del dispatcher (*allocatore*), sceglierne uno, solitamente attribuendo a turno a ciascun processo in coda un po' di tempo di esecuzione. Tale strategia è nota come tecnica round-robin; peraltro, si possono anche usare dei livelli di priorità.

La coda a lungo termine è una lista di nuovi job in attesa di utilizzare il sistema. Il sistema operativo aggiunge job al sistema trasferendo un processo dalla coda a lungo termine a quella a breve termine; contemporaneamente, dev'essere allocata per il processo entrante una porzione di memoria principale. Quindi, il sistema deve accertarsi di non sovraimpegnare la memoria né l'elaborazione, ammettendo troppi processi nel sistema. Esiste poi una coda per ciascun dispositivo di I/O, in quanto diversi processi possono richiederne l'uso contemporaneamente. Tutti i processi in attesa di utilizzare un dispositivo sono allineati nella coda di tale dispositivo: anche in questo caso, è compito del sistema operativo scegliere il processo cui assegnare un dispositivo di I/O disponibile.

In caso di interruzione, il sistema operativo, o meglio, quella parte di esso che costituisce il gestore delle interruzioni, riceve il controllo del processore. Un processo può chiamare un particolare servizio del sistema operativo, ad esempio un gestore di un dispositivo di I/O, attraverso la chiamata di un servizio. In questo caso, il punto di entrata nel sistema operativo è il gestore delle chiamate di servizio. Comunque, una volta che l'interruzione o il servizio sono stati gestiti, viene chiamato lo schedulatore a breve termine per scegliere un processo da eseguire.

Quella fin qui condotta è una descrizione funzionale; i dettagli e l'architettura modulare di questa parte del sistema operativo saranno notevolmente diversi nei vari sistemi. Buona parte della ricerca tesa allo sviluppo dei sistemi operativi si è focalizzata sulla scelta di algoritmi e strutture dati per quelle funzioni che forniscono equità, tempi di riposta differenziali ed efficienza.

Struttura del sistema

Con l'aggiunta di nuove caratteristiche, e mentre l'hardware sottostante diventava sempre più complesso e versatile, sono aumentate anche le dimensioni e la complessità dei sistemi operativi. CTSS, reso operativo al MIT nel 1963, si componeva approssimativamente di 32000 parole di memoria a 36 bit. OS/360, introdotto un anno più tardi dall'IBM, possedeva più di un milione di istruzioni macchina. Nel 1975 il sistema Multics, sviluppato dal MIT e dai Bell Labs, è cresciuto fino a superare i venti milioni di istruzioni. È vero che, più di recente, sono stati in effetti introdotti alcuni sistemi operativi meno complessi per sistemi più piccoli, tuttavia anche questi sono divenuti sempre più articolati, man mano che l'hardware sottostante si sviluppava e crescevano le richieste degli utenti. Quindi, l'UNIX di oggi è molto più complesso di quella sorta di sistema-giocattolo sviluppato da pochi programmatori di talento nei primi anni '70, e il semplice MS-DOS ha aperto la via a sistemi ricchi e articolati, come OS/2 e Windows 95.

Le dimensioni di un sistema operativo pienamente sviluppato e le ardue difficoltà che esso comporta, hanno causato tre problemi, deprecabili ma del tutto comuni. Primo: la comparsa di nuovi sistemi operativi soffre, rispetto alla domanda, di un cronico ritardo, che caratterizza anche gli aggiornamenti di sistemi già noti. Secondo: i sistemi hanno bachi latenti, che si mostrano

Tabella 2.4 *Architettura di un sistema operativo gerarchico*

Livello	Nome	Oggetti	Operazioni di esempio
13	Shell	Ambiente di programmazione utente	Comandi nel linguaggio di shell
12	Processi utente	Processi utente	Quit, kill, suspend, resume
11	Directory	Directory	Create, destroy, attach, detach, search, list
10	Dispositivi	Dispositivi esterni, come stampanti, schermi e tastiere	Create, destroy, open, close, read, write
9	File system	File	Create, destroy, open, close, read, write
8	Comunicazioni	Pipe	Create, destroy, open, close, read, write
7	Memoria virtuale	Segmenti, pagine	Read, write, fetch
6	Memoria secondaria locale	Blocchi di dati, canali dei dispositivi	Read, write, allocate, free
5	Processi primitivi	Processi primitivi, semafori, lista dei processi pronti	Suspend, resume, wait signal
4	Interruzioni	Programmi per la gestione delle interruzioni	Invoke, mask, unmask, retry
3	Procedure	Procedure, stack delle chiamate, display	Mark stack, call, return
2	Insieme delle istruzioni	Stack di valutazione, interprete di microprogramma, dati scalari e array	Load, store, add, subtract, branch
1	Circuiti elettronici	Registri, porte, bus, ecc.	Clear, transfer, ectivate, complement

solo sul campo e che bisogna poi eliminare. Terzo: le prestazioni spesso non sono all'altezza delle attese.

Per gestire la complessità dei sistemi operativi e risolvere questi problemi, la ricerca si è concentrata, nel corso degli anni, sulla struttura del software. Determinate sue caratteristiche appaiono ovvie: il software dev'essere modulare, per favorire l'organizzazione dello sviluppo del software e circoscrivere la diagnosi e la correzione degli errori. I moduli devono possedere interfacce ben definite e più semplici possibile, per facilitare l'attività di programmazione ed

agevolare l'evoluzione del sistema: infatti, se dotato di interfaccia pulita e minimale, un modulo può essere sostituito con un impatto minimo sugli altri.

Tuttavia, per grandi sistemi operativi, che vanno da centinaia di migliaia a milioni di linee di codice, la sola programmazione modulare non si è rivelata sufficiente e si è fatto ricorso, in misura sempre crescente, ai concetti di strati gerarchici e astrazione dell'informazione. La strottura gerarchica di un sistema operativo modemo separa le sue funzioni a seconda della loro complessità, della scala temporale caratteristica e del livello di astrazione. Si può vedere un sistema come una serie di livelli: ciascuno di essi esegue un sottoinsieme correlato delle funzioni necessarie al sistema operativo, ed è in relazione con il successivo livello più basso per eseguire funzioni più primitive e per nasconderne i dettagli, e fornisce servizi allo strato successivo di più alto livello. In teoria, bisognerebbe definire i livelli in modo che cambiamenti in un livello non richiedano cambiamenti negli altri; pertanto un problema è stato scomposto in un insieme di sottoproblemi, maggiormente gestibili.

In generale, gli strati più bassi trattano eventi di durata assai più ridotta; alcune parti del sistema operativo interagiscono infatti direttamente con l'hardware, dove gli eventi possono avere una scala temporale dell'ordine di pochi miliardesimi di secondo. All'altro estremo, invece, altre parti del sistema operativo comunicano con l'utente, che fornisce comandi a un ritmo molto più lento, all'incirca uno ogni pochi secondi. L'uso di un insieme di livelli si adatta bene a quest'ambiente.

L'applicazione di questi principi varia grandemente nei diversi sistemi operativi contemporanei: a questo punto, al fine di acquisire una conoscenza sommaria dei sistemi operativi, è utile presentare un modello di un sistema operativo gerarchico, ad esempio quello contenuto in [BROW84] e [DENN84], benché non corrisponda ad alcun sistema operativo realmente esistente. Il sistema modello è definito nella Tabella 2.4 ed è costituito dai seguenti livelli:

- **Livello 1**: contiene i circuiti elettronici; i suoi oggetti caratteristici sono registri, celle di memoria e porte logiche. Le operazioni definite su questi oggetti sono azioni del tipo: pulire un registro o leggere una locazione di memoria.
- **Livello 2**: l'insieme delle istruzioni del processore; a questo livello le operazioni sono quelle permesse dall'insieme di istruzioni del linguaggio macchina: ad esempio, add (*addiziona*), subtract (*sottrai*), load (*carica*) e store (*memorizza*).
- **Livello 3**: aggiunge il concetto di procedura o subroutine e le operazioni di chiamata e ritorno.
- **Livello 4**: introduce le interruzioni, che determinano il salvataggio del contesto corrente e chiamano una routine di gestione dell'interrupt.

Questi primi quattro livelli non sono parte del sistema operativo, ma costituiscono l'hardware del processore. Tuttavia, già a questi livelli compaiono alcuni elementi del sistema operativo, ad esempio le routine di gestione degli interrupt. Al livello 5 raggiungiamo il sistema operativo vero e proprio e compaiono i concetti associati con la multiprogrammazione.

- **Livello 5**: viene introdotta la nozione di processo come programma in esecuzione. Fra i requisiti fondamentali di un sistema operativo, che supporti molti processi, è compresa la

capacità di sospendere e ripristinare i processi stessi. Ciò richiede il salvataggio dei registri hardware, in modo che l'esecuzione possa passare da un processo ad un altro. Inoltre, se i processi devono cooperare, si rende necessario un meccanismo di sincronizzazione: una delle tecniche di segnalazione più semplici è il semaforo. Per lo studio di questo concetto essenziale nella progettazione dei sistemi operativi, si rimanda al Capitolo 5.

- **Livello 6**: dispositivi di memoria secondaria del computer. A questo livello avvengo le funzioni di posizionamento delle testine di lettura/scrittura ed il trasferimento vero e proprio dei blocchi di dati. Il livello 6 è in relazione col precedente per schedulare le operazioni e per notificare al processo richiedente il completamento di un'operazione. I livelli più alti si occupano degli indirizzi su disco e provvedono a richiedere il blocco appropriato al driver del dispositivo, al livello 5.

- **Livello 7**: crea uno spazio di indirizzamento logico per i processi. Questo livello organizza lo spazio degli indirizzi virtuali in blocchi che possono essere trasferiti fra la memoria principale e la memoria secondaria. Si usano comunemente tre schemi: quelli che usano pagine di dimensione fissa, quelli che usano segmenti di lunghezza variabile e quelli che li usano entrambi. Quando un blocco di cui si ha bisogno non è nella memoria principale, la logica di questo livello richiede un trasferimento al livello 6.

Fino a questo punto il sistema operativo utilizza le risorse di un singolo processore. A partire dal livello 8, tratta invece oggetti esterni, come dispositivi periferici ed eventualmente reti e computer connessi alle reti. A questi livelli superiori troviamo oggetti logici etichettati da nomi, che possono essere condivisi fra i processi su un unico computer o su diverse macchine.

- **Livello 8**: comunica informazioni e messaggi tra i processi. Mentre il livello 5 fornisce un meccanismo primitivo di gestione dei segnali per la sincronizzazione dei processi, questo livello tratta una condivisione dell'informazione più ricca. A questo scopo, uno degli strumenti più potenti è il pipe, un canale logico per il flusso dei dati fra i processi. Un pipe è definito dal suo output da un processo e dal suo input in un altro processo; può anche essere usato per collegare dispositivi esterni o file ai processi. Di questo problema si occuperà il Capitolo 6.

- **Livello 9**: supporta la memorizzazione a lungo termine di file con nome. A questo livello i dati sulla memoria secondaria sono visti in termini di entità astratte di lunghezza variabile. Ciò è in contrasto con il punto di vista orientato all'hardware, in termini di tracce, settori e blocchi di dimensione fissa espresso dal livello 6.

- **Livello 10**: fornisce l'accesso ai dispositivi esterni tramite interfacce standardizzate.

- **Livello 11**: gestisce l'associazione fra gli identificatori interni ed esterni degli oggetti e delle risorse di sistema. L'identificatore esterno è un nome che può essere utilizzato da un'applicazione o da un utente; l'identificatore interno è un indirizzo o un altro indicatore che può essere usato dai livelli più bassi del sistema operativo per localizzare o controllare un oggetto. Queste associazioni sono mantenute in una directory, le cui entry comprendono non solo il mapping identificatori esterni/interni, ma anche altre caratteristiche, come i diritti di accesso.

- **Livello 12**: fornisce una funzionalità pienamente sviluppata per il supporto dei processi, che va ben oltre ciò che è offerto al livello 5. Al livello 5, sono mantenuti solo i contenuti dei registri del processore associati con un processo, oltre alla logica per gestire l'alternarsi dei processi. Al livello 12, invece, viene supportata tutta l'informazione necessaria per la gestione ordinata dei processi. Essa comprende lo spazio di indirizzamento virtuale dei processi, una lista degli oggetti e dei processi con cui il processo può interagire e i vincoli di quell'interazione, i parametri passati al processo all'atto della sua creazione e ogni altra caratteristica dei processi che possa essere usata dal sistema operativo per controllarli.
- **Livello 13**: fornisce all'utente un'interfaccia verso il sistema operativo, definita shell perché separa l'utente dai dettagli del sistema operativo, che si presenta semplicemente come una collezione di servizi. La shell accetta i comandi utente o le istruzioni per il controllo dei job, li interpreta, e crea e controlla i processi a seconda delle necessità.

Questo modello ipotetico di un sistema operativo fornisce una descrizione utile e serve come linea guida per l'implementazione. Andando avanti nello studio, il lettore può tornare a riferirsi a questa struttura per avere un quadro del contesto di ogni particolare problema dell'architettura oggetto di discussione.

2.4 Caratteristiche dei sistemi operativi moderni

Nel corso del tempo c'è stata un'evoluzione graduale della struttura e delle capacità dei sistemi operativi. Negli ultimi anni sono stati introdotti alcuni elementi architetturali nuovi, sia nei nuovi sistemi operativi, sia negli aggiornamenti di sistemi già esistenti, introducendo così modifiche rilevanti. Tali sistemi operativi moderni rappresentano una risposta agli sviluppi dell'hardware e alle nuove applicazioni. Tra i dispositivi hardware principali, ricordiamo le macchine multiprocessore, che aumentano in modo rilevante la velocità di computazione, le connessioni di rete ad alta velocità, l'aumento in dimensioni e in varietà dei dispositivi di memorizzazione. In campo applicativo, l'architettura dei sistemi operativi è stata influenzata dalle applicazioni multimediali, da Internet e dal Web.

La quantità di cambiamenti richiesti ai sistemi operativi, impone non solo di modificare e migliorare le architetture esistenti, ma anche di trovare nuove modalità di organizzazione. Sono stati sperimentati molti diversi approcci e diversi elementi architetturali, sia nei sistemi operativi sperimentali che in quelli commerciali, ma l'impegno maggiore ha riguardato i seguenti ambiti:

- Architettura microkernel
- Multithread
- Multiprocessing simmetrico
- Sistemi operativi distribuiti
- Architettura orientata agli oggetti.

Gran parte dei sistemi operativi, fino a poco tempo fa, presentava un grande kernel monolitico che forniva la maggior parte delle funzionalità, tra cui la schedulazione, il file system, le funzioni di rete, i driver dei dispositivi, la gestione della memoria ed altre ancora. Un'**architettura microkernel**, invece, assegna al kernel solo poche funzioni essenziali, come la gestione dei diversi spazi di indirizzamento, la comunicazione fra i processi (IPC) e la schedulazione di base. Gli altri servizi del sistema operativo sono forniti da processi, talvolta chiamati server, eseguiti in modalità utente e trattati dal microkernel come tutte le altre applicazioni. Questo approccio separa lo sviluppo del kernel da quello dei server, che possono essere adattati ad applicazioni specifiche o alle necessità dell'ambiente operativo. L'approccio microkernel semplifica l'implementazione, aumenta la flessibilità ed è appropriato per un ambiente distribuito. In sintesi, un microkernel interagisce con processi server locali e remoti allo stesso modo, agevolando la costruzione di sistemi distribuiti.

Il **multithread** è una tecnica per mezzo della quale un processo, eseguendo un'applicazione, viene suddiviso in thread (*linee di esecuzione separate*), eseguibili simultaneamente. È possibile operare la seguente distinzione:

- **Thread**: l'unità di allocazione del lavoro. Viene eseguito sequenzialmente, e lo si può interrompere in modo che il processore passi ad un altro thread. Dal punto di vista della schedulazione e dell'allocazione, questo concetto è equivalente a quello di processo nella maggior parte dei sistemi operativi.
- **Processo**: una collezione di uno o più thread e delle risorse di sistema associate (ad esempio, la memoria, i file aperti e i dispositivi), che corrisponde da vicino al concetto di programma in esecuzione. Separando una singola applicazione in molti thread, il programmatore controlla efficacemente la modularità dell'applicazione e la temporizzazione degli eventi ad essa correlati. Dal punto di vista del job o dell'applicazione, questo concetto equivale a quello di processo nella maggioranza degli altri sistemi operativi.

Il multithread è utile per quelle applicazioni che effettuano un insieme di compiti essenzialmente indipendenti, che non necessitano di essere serializzati. Un esempio è un server di database che risponde a diverse richieste dei client e le elabora. Se diversi thread sono in esecuzione all'interno di uno stesso processo, il passaggio da un thread all'altro richiede un sovraccarico minore rispetto al passaggio da un processo ad un altro. I thread servono anche a strutturare i processi che fanno parte del kernel, come si vedrà nei prossimi capitoli.

Fino a poco tempo fa, praticamente tutti i personal computer a singolo utente e le workstation contenevano un solo microprocessore general-purpose. Mentre cresceva la richiesta di prestazioni, e diminuiva il costo dei microprocessori, la situazione è cambiata, grazie alla comparsa di sistemi multimicroprocessore. Il **multiprocessore simmetrico (SMP)** rappresenta una soluzione valida per ottenere il massimo di efficienza ed affidabilità. Il termine SMP si riferisce sia all'architettura hardware del computer sia al comportamento del sistema operativo al di sopra di tale architettura. Un multiprocessore simmetrico può essere definito come un computer singolo dotato delle seguenti caratteristiche:

1. Sono presenti molti processori.

2. Questi processori condividono la stessa memoria principale e le stesse funzionalità di I/O, e sono interconnessi da un bus o da un altro schema interno di connessione.
3. Tutti i processori possono effettuare le stesse funzioni (da cui il termine *simmetrico*).

Il sistema operativo di un SMP schedula i processi o i thread su tutti i processori. Rispetto ad un'architettura monoprocessore, SMP presenta potenzialmente diversi vantaggi, tra cui i seguenti:

- **Prestazioni**: se il lavoro da svolgere può essere organizzato in modo da eseguirne in parallelo alcune porzioni, un sistema con molti processori otterrà migliori risultati rispetto ad un sistema dello stesso tipo a singolo processore (Figura 2.12).
- **Disponibilità:** in un multiprocessore simmetrico tutti i processori possono svolgere le medesime funzioni, perciò il fallimento di uno di essi non ferma la macchina, e il sistema può continuare a funzionare anche se con prestazioni ridotte.
- **Crescita incrementale:** un utente può aumentare le prestazioni di un sistema aggiungendo processori addizionali.
- **Scalabilità:** è possibile offrire una gamma di prodotti con caratteristiche di prezzo e di prestazioni diverse, in base al numero di processori configurati nel sistema.

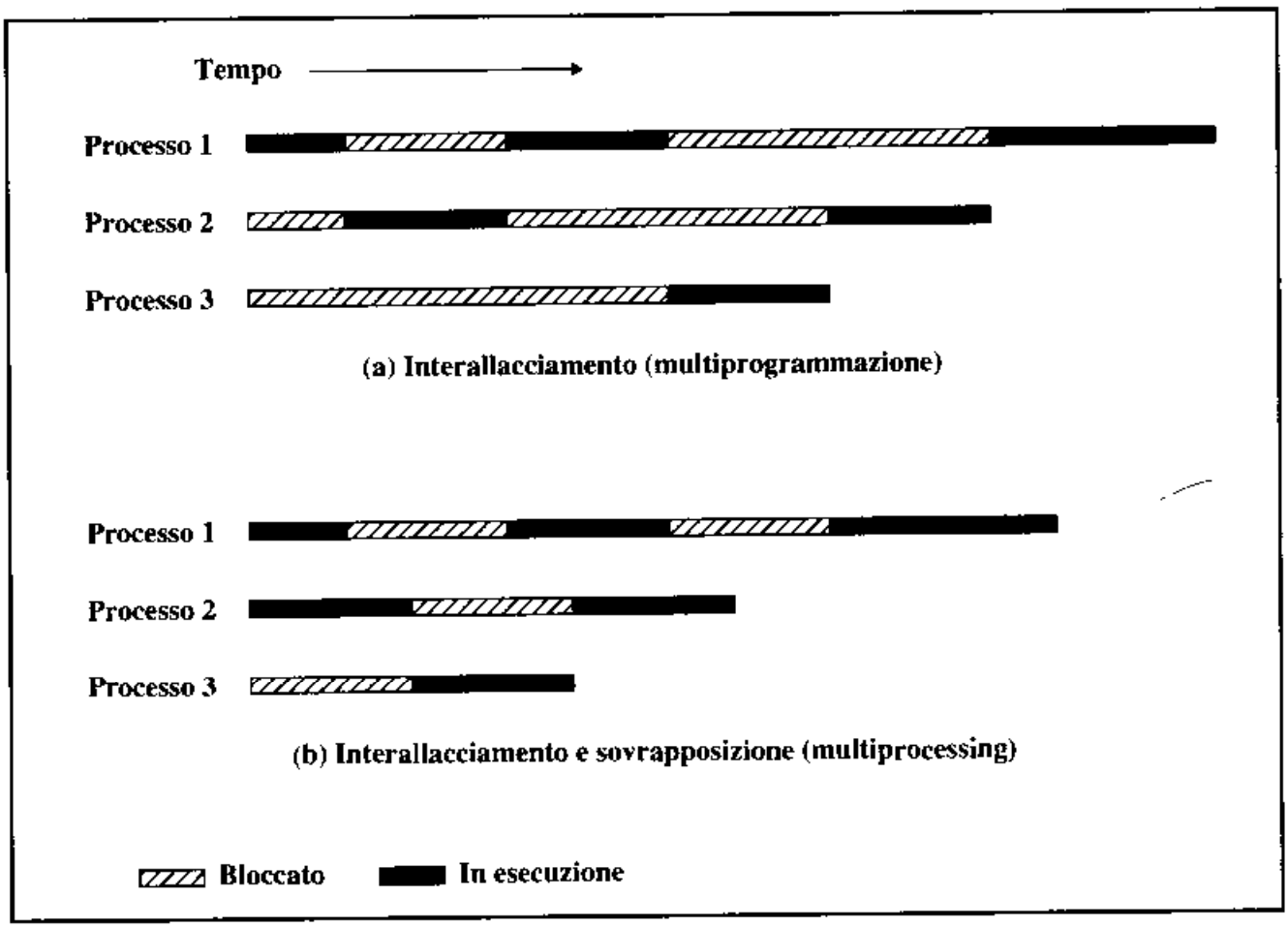


Figura 2.12 *Multiprogrammazione ed elaborazione multiprocessing*

Bisogna però osservare che si tratta di benefici potenziali, piuttosto che garantiti: il sistema operativo deve fornire strumenti e funzioni per sfruttare il parallelismo in un sistema SMP.

Il multithread spesso viene esaminato insieme con SMP, benché si tratti di due funzionalità indipendenti. Il multithread è utile per strutturare le applicazioni ed i processi del kernel, anche in macchine monoprocessore, mentre una macchina SMP si può usare per processi senza thread, poiché diversi processi possono essere eseguiti in parallelo. In ogni caso le due funzionalità sono complementari e si possono efficientemente utilizzare insieme.

Una caratteristica interessante di SMP sta nel fatto che l'esistenza di molti processori è trasparente all'utente. Il sistema operativo si preoccupa della schedulazione dei thread o dei processi nei singoli processori e della sincronizzazione fra i processori stessi. Questo testo esamina i meccanismi di schedulazione e sincronizzazione utilizzati affinché l'utente possa lavorare come su un sistema singolo. Un problema diverso è fornire l'aspetto di un singolo sistema ad un cluster di computer separati, cioè ad un sistema multicomputer. Si dispone in questo caso di una serie di entità (i computer), ciascuna delle quali possiede la sua propria memoria principale, la memoria secondaria ed altri moduli di I/O. Un **sistema operativo distribuito** offre l'illusione di un singolo spazio di memoria principale e di memoria secondaria, più altre funzionalità per un accesso unificato alle diverse risorse, come un file system distribuito. Benché i cluster vadano sempre più diffondendosi, ed esistano in commercio molti prodotti di questo tipo, lo stato dell'arte per i sistemi operativi distribuiti è meno avanzato rispetto a quello dei sistemi monoprocessore e SMP. (Per questi sistemi, si rimanda alla Parte Sesta).

L'innovazione più recente nell'architettura dei sistemi operativi è l'introduzione della **progettazione orientata agli oggetti**, che permette di disciplinare il processo di espansione modulare dei kernel di piccole dimensioni; una struttura basata sugli oggetti consente ai programmatori di modificare un sistema operativo senza distruggere l'integrità del sistema. Lo sviluppo orientato agli oggetti facilita la scrittura di strumenti distribuiti e di veri e propri sistemi operativi distribuiti.

2.5 Panornmicn su Windows NT

Questo testo si propone di far conoscere i principi architetturali e le caratteristiche implementative dei sistemi operativi di oggi, quindi una trattazione puramente concettuale o teorica risulterebbe inadeguata. Per evidenziare i concetti di fondo, collegandoli però a delle possibili scelte progettuali concrete, sono stati scelti come oggetto di analisi due sistemi operativi reali:

- **Windows NT**: sistema operativo a singolo utente e multitasking, progettato per essere eseguito su diversi PC e workstation. È uno dei pochi sistemi operativi commerciali recenti che sia stato progettato praticamente da zero, ed è per questo che si è potuto avvalere di tutti i più recenti sviluppi della tecnologia nell'ambito dei sistemi operativi.
- **UNIX**: sistema operativo multitasking, in origine destinato a minicomputer, ma implementato su un'ampia tipologia di macchine, dai potenti microcomputer fino ai supercomputer.

Questa sezione è dedicata a Windows NT, mentre UNIX verrà descritto in quella successiva.

Storia

L'antenato più lontano di Windows NT è un sistema operativo sviluppato dalla Microsoft per il primo personal computer IBM, e cioè MS-DOS o PC-DOS. La prima versione, DOS 1.0, si componeva di 4000 linee di codice sorgente in linguaggio Assembler e richiedeva 8KB di memoria su di un microprocessore Intel 8086, e fu presentata nell'agosto del 1981.

Quando IBM realizzò un personal computer dotato di hard disk, il PC XT, Microsoft sviluppò DOS 2.0 (1983), per supportare l'hard disk e fornire directory gerarchiche. In precedenza, i dischi potevano contenere solo una directory, contenente al massimo 64 file. Se tale caratteristica poteva essere adeguata nell'era dei floppy disk, appariva però troppo limitante per un hard disk, e la restrizione ad una singola directory era troppo scomoda. La nuova versione consentiva alle directory di contenere sia sottodirectory, sia file; ampliava di molto l'insieme dei comandi incorporati nel sistema operativo, e metteva a disposizione funzioni che nella versione 1 erano realizzate da programmi esterni forniti come utility. Tra le nuove caratteristiche, alcune erano simili a quelle di UNIX, come la redirezione dell'I/O (cioè la possibilità per una data applicazione di cambiare l'identità dell'input e dell'output) e la stampa in background. La porzione residente in memoria aumentò fino a 24KB.

Quando, nel 1984, IBM annunciò il PC AT, Microsoft introdusse DOS 3.0. Il computer AT conteneva un processore Intel 80286, dotato di indirizzamento esteso e capacità di protezione della memoria, caratteristiche, queste, non utilizzate dal DOS. Per rimanere compatibile con le precedenti versioni, il sistema operativo usava l'80286 semplicemente come un "8086 veloce", ma supportava la nuova tastiera e l'hard disk, raggiungendo così un'occupazione di memoria di 36KB. Vi furono poi miglioramenti successivi alla versione 3.0: la versione DOS 3.1, presentata nel 1984, conteneva il supporto per reti di PC, senza cambiare la dimensione della memoria residente (ciò fu reso possibile dall'incremento della quantità di sistema operativo che doveva essere trasferito fra la memoria ed il disco tramite swapping). DOS 3.3, presentato nel 1987, forniva supporto alla nuova linea di macchine IBM, i PS/2. Nemmeno questa versione sfruttava le capacità dei processori dei PS/2, dotate di chip 80286 e di 80386 a 32 bit. La porzione residente in memoria cresceva fino ad almeno 46KB, arrivando anche a dimensioni maggiori quando venivano selezionate determinate estensioni facoltative.

Si continuava dunque ad usare DOS in un ambiente hardware ben superiore alle sue capacità, a maggior ragione dopo l'introduzione dell'80486 e del chip Intel Pentium, che avevano reso disponibili potenza e caratteristiche che però non potevano essere sfruttate da un sistema tanto elementare. Nel frattempo, nei primi anni '80, Microsoft iniziò a sviluppare un'interfaccia utente grafica (GUI, *Graphical User Interface*), da interporre fra l'utente ed il DOS: scopo di Microsoft era competere con Macintosh, il cui sistema operativo restava ineguagliato per facilità d'uso. Nel 1990 Microsoft lanciò una versione di GUI, nota come Windows 3.0, la cui semplicità d'uso si avvicinava a quella del Macintosh, ma appariva ancora limitata dal fatto di dover essere eseguita insieme con il sottostante DOS.

Microsoft, dopo un tentativo fallito di sviluppare con IBM una nuova generazione di sistemi operativi volti a sfruttare la potenza dei nuovi microprocessori e dotati della facilità d'uso di

Windows, ha cominciato a sviluppare in proprio Windows NT. Tale sistema sfrutta la potenza dei microprocessori moderni e fornisce un multitasking completo in un ambiente a singolo utente. Tuttavia IBM ha continuato a sviluppare OS/2 per conto proprio. Come Windows NT, OS/2 è un sistema operativo multitasking, multithreading a singolo utente.

La prima versione di Windows NT (3.1), dotata della stessa GUI di Windows 3.1, risale al 1993. Comunque NT 3.1 era un sistema operativo a 32 bit completamente nuovo, in grado di supportare le vecchie applicazioni DOS e Windows, come di fornire supporto alle applicazioni OS/2.

Dopo diverse versioni di NT 3.x, Microsoft ha presentato NT 4.0, che essenzialmente ne riprende l'architettura; il cambiamento esterno più rilevante sta nel fatto che presenta la stessa interfaccia di Windows 95, mentre, quanto all'architettura, diverse componenti grafiche, eseguite in modalità utente come parte del sottosistema Win32 in 3.x, sono state trasportate in Windows NT Executive, eseguito in modalità kernel. Il vantaggio di questo cambiamento sta nell'accelerazione delle operazioni di queste importanti funzioni; come principale controindicazione, si rileva che queste funzioni grafiche hanno così accesso a servizi di sistema di basso livello, il che potrebbe avere un impatto sull'affidabilità del sistema operativo.

Nel 1998 Microsoft ha introdotto un nuovo importante aggiornamento, NT5.0, dove la componente Executive e l'architettura microkernel sono fondamentalmente le stesse di NT4.0, con alcune nuove caratteristiche. Il fatto più rilevante, in 5.0, è l'aggiunta di servizi e funzioni per supportare l'elaborazione distribuita. L'innovazione più importante di 5.0 è Active Directory, un servizio distribuito di directory in grado di collegare i nomi di oggetti arbitrari a diversi tipi di informazione riguardo quegli oggetti.

La maggior parte dei riferimenti a NT contenuti in questo libro (a meno di diverse indicazioni) è basata sulla versione 4.0.

Da ultimo, resta da esaminare la distinzione fra NT Server e NT Workstation. La Tabella 2.5 riassume le principali differenze funzionali: in sostanza, il microkernel, l'architettura ed i servizi di Executive rimangono sempre gli stessi, ma NT Server comprende alcuni servizi necessari per usare NT come server di rete.

Multitasking a singolo utente

Windows NT, insieme con OS/2 e Mac/OS, è forse il più importante esempio delle nuove tendenze nello sviluppo dei sistemi operativi per personal computer. Windows NT intende rispondere alla necessità di sfruttare l'enorme potenza dei microprocessori odierni a 32 bit, in grado di competere in velocità, sofisticazione dell'hardware e capacità di memoria con i mainframe e i minicomputer di pochi anni fa.

Benché il fine di questi nuovi sistemi operativi sia ancora quello di supportare un singolo utente interattivo, si distinguono per essere sistemi multitasking. Soprattutto due fattori, nell'ambito dei personal computer, hanno stimolato l'introduzione del multitasking. Il primo fattore, dovuto alla crescente velocità e capacità di memoria dei microprocessori, che supportano la memoria virtuale, è la possibilità di sviluppare applicazioni più complesse e correlate. Ad esempio, un utente vuole produrre un documento, impiegando simultaneamente un elaboratore di testi, un programma di disegno e un foglio elettronico. Se egli desidera creare un disegno e incollarlo in un documento di testo, senza multitasking deve compiere i passi seguenti:

Tabella 2.5 *Differenze fra NT4.0 Workstation e NT4.0 Server*

	Workstation	Server
Connessione ad altri client	10	Illimitate
Connessione ad altre reti	Illimitate	Illimitate
Multiprocessing	2 processori	4 processori
Servizi di Accesso Remoto (RAS)	1 connessione	255 connessioni
Replicazione di directory	Importazione	Importazione ed esportazione
Servizi Macintosh	No	Sì
Validazione Logon	No	Sì
Fault del disco	No	Sì
Rete	Peer to peer	Server

1. Aprire il programma di disegno.
2. Creare il disegno e salvarlo in un file o in una clipboard temporanea.
3. Chiudere il programma di disegno.
4. Aprire il programma per l'elaborazione del testo.
5. Inserire il disegno nella posizione corretta.

Dovendo operare dei cambiamenti, l'utente deve chiudere il programma di scrittura, aprire quello per il disegno, modificare l'immagine grafica, salvarla, chiudere il programma di disegno, aprire quello di scrittura e inserire l'immagine aggiornata, con un prucedimento che in breve diventa fastidioso. Man mano che i servizi e le potenzialità a disposizione divengono più potenti e vari, l'ambiente a singolo task si rivela, infatti, sempre più scomodo e meno amichevole. In un ambiente multitasking, l'utente apre le applicazioni necessarie *e le lascia aperte*, e può agevolmente spostare l'informazione fra le diverse applicazioni, che hanno una o più finestre aperte e un'interfaccia grafica con un dispositivo di puntamento (mouse), per agevolare gli spostamenti nell'ambiente di lavoro.

Una seconda motivazione a favore del multitasking è la crescita dell'elaborazione client/server, per mezzo della quale un personal computer o una workstation (client) e un sistema host (server) sono utilizzati insieme per portare a termine una particolare applicazione. I due componenti sono collegati, e a ciascuno è assegnata quella parte di lavoro adatta alle sue capacità. L'elaborazione client/server può essere effettuata in una rete locale di personal computer e server, oppure attraverso un collegamento fra un sistema utente ed un host di grandi dimensioni, ad esempio un mainframe. Un'applicazione può coinvolgere uno o più personal computer ed uno o più server: per fornire la necessaria prontezza di risposta, il sistema operativo deve supportare un hardware sofisticato per la comunicazione in tempo reale, i protocolli di comunicazione associati e architetture per il trasferimento dei dati, gestendo in contemporanea l'interazione con l'utente.

Architettura

La Figura 2.13 illustra la struttura modulare di Windows NT, che lo rende particolarmente flessibile. NT è eseguibile su diverse piattaforme hardware e supporta anche applicazioni scritte per altri sistemi operativi.

Al pari di quasi tutti i sistemi operativi, NT separa il software applicativo da quello di sistema, noto come NT Executive, che è eseguito in modo kernel. Il software in modo kernel ha accesso ai dati di sistema ed all'hardware; il software rimanente, eseguito in modo utente, ha accesso limitato ai dati di sistema.

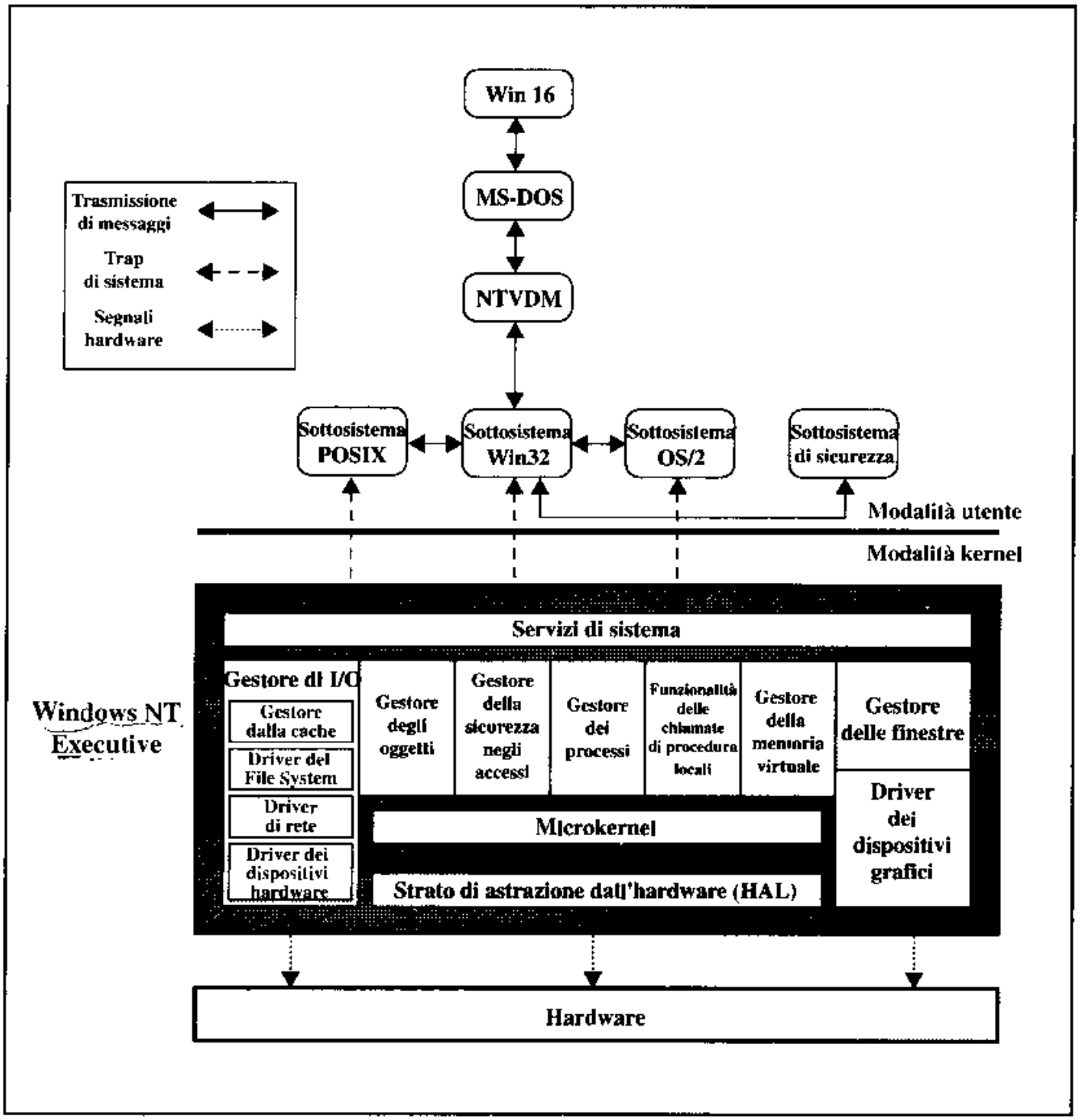


Figura 2.13 *Architettura di Windows NT 4.0*

NT Executive

Benché NT Executive comprenda un microkernel, non si tratta di un'architettura microkernel pura, ma di ciò che Microsoft definisce *architettura microkernel modificata;* tuttavia, come nel caso di un'architettura microkernel pura, NT è altamente modulare. Ciascuna funzione di sistema è gestita da una componente del sistema operativo; il resto del sistema operativo e tutte le applicazioni accedono a quella funzione attraverso tale componente, utilizzando un'interfaccia standardizzata, mentre ai dati chiave di sistema è possibile accedere solo attraverso la funzione appropriata. In linea di principio, ogni modulo può essere rimosso, aggiornato o rimpiazzato senza riscrivere l'intero sistema o le sue interfacce standard per la programmazione delle applicazioni (API). Comunque, a differenza di un sistema microkernel puro, al fine di migliorare le prestazioni NT è configurato in modo che molte delle funzioni di sistema fuori dal microkernel siano eseguite in modo kernel. I progettisti di NT hanno constatato che, con l'approccio microkernel puro, molte funzioni non microkernel richiedono diversi cambi di processo o di thread, cambi di modo e l'utilizzo d'ulteriori buffer di memoria. Resta da vedere se questa conclusione sia inevitabile e se sarà applicata da altri sistemi operativi commerciali.

Uno degli obiettivi nella progettazione di NT è la portabilità, cioè la capacità di essere eseguito non solo su macchine Intel, ma su una varietà di piattaforme hardware. Per soddisfare quest'obiettivo, la maggior parte di NT Executive ha una visione unificata dell'hardware sottostante, utilizzando la seguente struttura stratificata:

- **Strato di astrazione dall'hardware (HAL)**: mette in corrispondenza i comandi e le risposte dell'hardware generico con quelli particolari di una specifica piattaforma, come un processore Pentium, PowerPC o Alpha. HAL fa in modo che tutti i bus di sistema della macchina, i controller di accesso diretto a memoria (DMA), i controller di interrupt, i timer di sistema e i moduli di memoria appaiano al kernel allo stesso modo. Fornisce anche il supporto necessario per il multiprocessing simmetrico (SMP), di cui si parlerà più avanti.
- **Microkernel**: contiene i componenti più utili e più importanti del sistema operativo. Il kernel gestisce la schedulazione dei thread e dei processi, i cambi di contesto, la gestione delle eccezioni e delle interruzioni, e la sincronizzazione dei sistemi multiprocessore. A differenza del resto di Executive e del livello utente, il codice proprio del microkernel non viene eseguito in thread, ed è perciò la sola parte del sistema operativo che non sia prerilasciabile o paginabile.
- **Servizi Executive**: comprendono vari moduli per funzioni specifiche di sistema. Tutti questi servizi, tranne il gestore di I/O, dei moduli grafici e delle finestre (descritti in seguito), per accedere all'hardware devono passare attraverso HAL.
- **Servizi di sistema**: forniscono un'interfaccia al software in modalità utente.

Passiamo ora ad una sintetica descrizione di ciascuno dei moduli di servizi di Executive:

- **Gestore di I/O**: elabora le richieste in ordine di priorità da una coda di I/O. L'I/O di NT verrà esaminato nel Capitolo 11.

- **Gestore degli oggetti**: applica regole uniformi per mantenere e determinare la sicurezza degli oggetti, e per assegnare loro un nome. Crea inoltre degli *handle* per gli oggetti, che si compongono di informazioni per il controllo dell'accesso e di un puntatore all'oggetto stesso. Degli oggetti di NT si occupa, più avanti, questa stessa sezione.
- **Gestore della sicurezza negli accessi**: responsabile dell'applicazione delle regole per la validazione dell'accesso e per la generazione del controllo. Il modello di NT orientato agli oggetti permette una visione coerente ed uniforme della sicurezza, coinvolgendo le entità fondamentali di Executive. Infatti, NT utilizza le stesse routine per la validazione dell'accesso e per il controllo di tutti gli oggetti protetti, compresi file, processi, spazi di indirizzamento e dispositivi di I/O. Della sicurezza di NT si parlerà nel Capitolo 15.
- **Gestore dei processi**: crea e cancella oggetti e gestisce gli oggetti processo e thread. La gestione dei processi e dei thread di NT verrà descritta nel Capitolo 4.
- **Chiamate di procedura locali (LPC)**: questa funzionalità applica relazioni client/server fra le applicazioni e i sottosistemi di Executive in un sistema singolo, in modo simile alle chiamate di procedura remote (RPC) utilizzate per l'elaborazione distribuita. Di LPC si occuperà più avanti questa stessa sezione.
- **Gestore della memoria virtuale**: mappa gli indirizzi virtuali dello spazio di indirizzi del processo nelle pagine fisiche della memoria del computer. La gestione della memoria virtuale di NT verrà descritta nel Capitolo 8.
- **Moduli grafici e delle finestre**: crea l'interfaccia orientata alle finestre e gestisce i dispositivi grafici.

Sottosistemi di ambiente

NT è strutturato per supportare applicazioni scritte per Windows NT, Windows 95 e diversi altri sistemi operativi. NT fornisce questo supporto attraverso un ambiente Executive compatto, che usa *sottosistemi di ambiente* protetti, in altre parole quelle parti di NT che interagiscono con l'utente finale. Ciascun sottosistema è un processo separato ed Executive protegge il suo spazio di indirizzamento da quello degli altri sottosistemi e delle altre applicazioni. Un sottosistema protetto è dotato di un'interfaccia utente grafica, o a linea di comando, che determina l'aspetto e il modo in cui l'utente percepisce il sistema operativo. Inoltre, ciascun sottosistema protetto fornisce l'interfaccia di programmazione per le applicazioni per quel particolare sistema operativo: ciò significa che applicazioni create per un particolare ambiente operativo, possono essere eseguite senza cambiamenti su NT, poiché l'interfaccia del sistema operativo che vedono è la stessa per cui sono state scritte. Ad esempio, le applicazioni basate su OS/2 possono essere eseguite sotto NT senza modifiche. Inoltre, poiché il sistema NT è esso stesso progettato per essere indipendente dalla piattaforma, utilizzando lo strato di astrazione dall'hardware, dovrebbe essere relativamente semplice portare sia i sottosistemi protetti sia le applicazioni che essi supportano da una piattaforma hardware a un'altra. In molti casi, è sufficiente una semplice ricompilazione.

Tabella 2.6 *Alcune aree coperte dall'API del sottosistema Win32 [RICH97]*

Atomi	Reti
Controlli dei processi figli	Pipe e mailslot
Manipolazione della clipboard	Stampa
Comunicazioni	Processi e thread
Console	Manipolazione del database Registry
Debugging	Risorse
Librerie a linking dinamico (DLL)	Sicurezza
Gestione degli eventi	Servizi
File	Gestione delle eccezioni strutturata
Primitive grafiche di disegno	Informazioni sul sistema
Input di tastiera e del mouse	Backup su nastro
Gestione della memoria	Tempo
Servizi multimodiali	Gestione delle finestre

Windows NT può fornire i seguenti sottosistemi:

- MS-DOS NT Virtual DOS Machine (NTVDM)
- Win16 NTVDM
- Sottosistema OS/2
- Sottosistema POSIX
- Sottosistema Win32.

Fra questi, il più importante è senz'altro Win32, che è l'API implementata sia su Windows NT, sia su Windows 95. Alcune delle caratteristiche di Win32 non sono disponibili in Windows 95, ma quelle implementate in Windows 95, sono identiche a quelle di NT. La Tabella 2.6 elenca alcune delle funzioni chiave che Win32 fornisce al programmatore.

Si noti la posizione centrale del sottosistema Win32 nella Figura 2.13. Tutti gli altri sottosistemi a livello di utente effettuano scambi di messaggi con Executive attraverso Win32. Win32 è l'ambiente nativo per Windows NT, e le altre API del sistema operativo sono mappate attraverso Win32 per essere eseguite in NT e per accedere ai servizi di NT Executive.

Modello client/server

Executive, i sottosistemi protetti e le applicazioni sono strutturate utilizzando il modello di elaborazione client/server, il modello tipico per l'elaborazione distribuita, di cui si tratterà nella Parte Sesta. Questa stessa architettura può essere utilizzata internamente ad un sistema singolo come nel caso di NT.

Ciascun sottosistema di ambiente e sottosistema di servizio di Executive è implementato

attraverso uno o più processi; ciascuno di questi attende che un cliente invii richieste ad uno dei suoi servizi (memoria, creazione di processi, schedulazione). Il client può essere un programma applicativo o un altro modulo del sistema operativo, che richiede un servizio mandando un messaggio. Il messaggio viene instradato da Executive al Server appropriato, che effettua l'operazione richiesta e ritorna i risultati o le informazioni di stato attraverso un altro messaggio, nuovamente instradato da Executive al client.

Fra i vantaggi dell'architettura client/server possiamo elencare i seguenti:

- Semplifica NT Executive, al cui interno è possibile costruire una varietà di API senza conflitti o duplicazioni; inoltre, è possibile aggiungere nuove API senza particolari difficoltà.
- Aumenta l'affidabilità: ciascun modulo di servizio di Executive viene eseguito come un processo separato, con la propria partizione di memoria protetta dagli altri moduli. Inoltre, i client non possono accedere direttamente all'hardware o modificare la memoria di Executive, e possono fallire senza determinare il crash o corrompere il resto del sistema operativo.
- Permette alle applicazioni di comunicare con Executive in modo uniforme attraverso chiamate di procedura locali, senza diminuire la flessibilità. Il procedimento di trasmissione dei messaggi è nascosto alle applicazioni client da funzioni stub, che sono segnaposto non eseguibili conservate in librerie a linking dinamico (DLL). Quando un'applicazione effettua una chiamata API ad un sottosistema di ambiente, lo stub nell'applicazione client impacchetta i parametri della chiamata e li spedisce, come messaggio, a un sottosistema server che implementa la chiamata.
- Fornisce una base naturale per l'elaborazione distribuita. Tipicamente, l'elaborazione distribuita fa uso di un modello client/server, con chiamate di procedura remota implementate attraverso client distribuiti, moduli server e lo scambio di messaggi fra client e server. Con NT, un server locale può passare un messaggio ad uno remoto perché elabori la richiesta per conto di applicazioni client locali. I client non hanno bisogno di sapere se una richiesta è servita in modo locale o remoto. Infatti, si può passare dinamicamente dall'una all'altra modalità, sulla base delle condizioni di carico attuali e dei cambiamenti dinamici della configurazione.

Thread e SMP

Il supporto per i thread e per il multiprocessing simmetrico (SMP), di cui abbiamo trattato nella Sezione 2.4, sono due caratteristiche importanti di Windows NT. [CUST93] elenca le seguenti caratteristiche di Windows NT che supportano i thread e SMP:

- Le routine del sistema operativo possono essere eseguite su ogni prucessore disponibile, e routine differenti possono essere eseguite simultaneamente su diversi processori.
- NT supporta l'uso di thread di esecuzione multipli entro un singolo processo, che possono essere eseguiti su diversi processori simultaneamente.
- I processi del server possono utilizzare thread multipli per rispondere simultaneamente alle richieste di più client.

- NT fornisce meccanismi convenienti per la condivisione dei dati e delle risorse e possibilità flessibili per la comunicazione fra i processi.

Oggetti di Windows NT

Windows NT utilizza a fondo i concetti della progettazione orientata agli oggetti, il che facilita sia la condivisione delle risorse e dei dati fra i processi, sia la protezione delle risorse da accessi non autorizzati. Tra i principali concetti orientati agli oggetti presenti in NT, si possono elencare i seguenti:

- **Incapsulamento**: un oggetto si compone di uno o più elementi di dati, chiamati attributi, e di una o più procedure, che possono manipolare quei dati, chiamate servizi; e il solo modo per accedere ai dati dell'oggetto sta nel chiamare uno dei servizi dell'oggetto stesso. In questo modo i dati si possono facilmente proteggere da usi non autorizzati o scorretti (ad esempio, cercare di eseguire un dato non eseguibile).
- **Classi degli oggetti e istanze**: la classe di un oggetto è uno schema che ne elenca gli attributi e i servizi, e definisce determinate sue caratteristiche. Il sistema operativo, a seconda delle necessità, può creare istanze specifiche di una classe di oggetti. Ad esempio, esiste una classe singola di oggetti per i processi, e un oggetto processo per ogni processo correntemente attivo. Questo approccio semplifica la creazione e la gestione degli oggetti.
- **Ereditarietà**: non è supportata a livello utente, ma lo è, entro certi limiti, all'interno di Executive. Ad esempio, gli oggetti Directory sono esempi di oggetti contenitore, in grado di trasmettere le loro proprietà agli oggetti in essi contenuti. Si supponga di avere una directory nel file system con il flag *compresso* settato: anche i file creati dentro il contenitore directory avranno il flag *compresso* settato.
- **Polimorfismo**: internamente NT usa un insieme comune di funzioni API per manipolare oggetti di diverso tipo, tuttavia NT non è completamente polimorfo, perché ci sono molte API specifiche, per tipi di oggetti specifici.

Il lettore che non ha familiarità con i concetti della programmazione orientata agli oggetti dovrebbe consultare, nella parte finale del volume, l'Appendice B.

Non tutte le entità di Windows NT sono oggetti: essi vengono utilizzati nei casi in cui i dati sono aperti all'accesso in modalità utente, o quando l'accesso ai dati è condiviso o limitato. I file, i processi, i thread, i semafori, i timer, le finestre sono tutti entità, rappresentate da oggetti. NT crea e gestisce tutti i tipi di oggetti in modo uniforme mediante il gestore degli oggetti, che è responsabile della loro creazione e distruzione per conto delle applicazioni, e garantisce l'accesso ai servizi e ai dati di un oggetto.

Ciascun oggetto all'interno di Executive, talvolta definito come oggetto kernel (per distinguerlo dagli oggetti a livello utente, di cui non è responsabile Executive), esiste come un blocco di memoria allocato e accessibile solo dal kernel. Alcuni elementi della struttura dati (tra cui il nome dell'oggetto, i parametri relativi alla sicurezza e i contatori di utilizzo), sono comuni a tutti i tipi di oggetti, mentre altri elementi sono specifici per un tipo particolare (ad esempio, la priorità di un oggetto thread). Queste strutture dati degli oggetti kernel sono accessibili solo dal

kernel stesso: un'applicazione non è in grado di localizzarle e leggervi o scrivervi direttamente, ma può manipolare gli oggetti indirettamente, per mezzo delle apposite funzioni, supportate da Executive. Quando un'applicazione richiede di creare un oggetto, riceve un *handle* per l'oggetto stesso; un handle è un puntatore all'oggetto referenziato, che può essere usato da qualsiasi thread entro il medesimo processo, per chiamare le funzioni di Win32 che operano sugli oggetti.

Gli oggetti contengono informazioni sulla sicurezza, sotto forma di un descrittore di sicurezza (SD), e possono utilizzarle per restringere l'accesso all'oggetto stesso. Ad esempio, un processo può creare un oggetto semaforo con nome, per far sì che solo certi utenti possano aprirlo e usarlo. Il descrittore di sicurezza per l'oggetto semaforo può elencare a quali utenti sia permesso, o negato, accedere al semaforo, e in che forma sia loro concesso di farlo (lettura, scrittura, modifica ecc.).

In NT, gli oggetti possono avere un nome oppure no. Quando un processo crea un oggetto senza nome, il gestore degli oggetti restituisce un handle a quell'oggetto, che è l'unico modo per riferirsi a esso. Se il nome dell'oggetto esiste, gli altri processi possono usarlo per ottenere un handle all'oggetto stesso. Ad esempio, se il processo A desidera sincronizzarsi con il processo B, può creare un oggetto evento con nome, e passare il nome dell'evento a B che, a sua volta, può aprire ed usare quell'oggetto evento. Comunque, se A desidera semplicemente usare l'evento per sincronizzare due propri thread fra loro, può creare un oggetto evento senza nome, non essendo necessario che altri processi usino quell'evento.

Come esempio di oggetti gestiti da NT, possiamo elencare le due categorie gestite dal microkernel:

- **Oggetti di controllo**: controllano le operazioni del microkernel in aree che non riguardano l'allocazione e la sincronizzazione. La Tabella 2.7 elenca gli oggetti di controllo del microkernel.

Tabella 2.7 *Oggetti di controllo del microkernel NT [MS96]*

Chiamate di procedura asincrone	Utilizzate per effettuare il break dell'esecuzione di un thread specificato e per fare in modo che una procedura sia chiamata in uno specifico modo del processore
Interruzioni	Utilizzate per connettere una sorgente di interruzione con una routine di servizio dell'interruzione attraverso un'entry nella Interrupt Dispatch Table (IDT). Ciascun processore ha un'IDT che viene utilizzata per smistare le interruzioni che avvengono in quel processore
Processi	Rappresentano lo spazio degli indirizzi virtuali e le informazioni di controllo necessarie per l'esecuzione di un insieme di oggetti thread. Un processo contiene un puntatore ad una mappa degli indirizzi, una lista dei thread pronti (contenente oggetti thread), una lista dei thread che appartengono al processo, il tempo totale accumulato da tutti i thread in esecuzione all'interno del processo e una priorità base.
Profili	Utilizzati per misurare la distribuzione del tempo di esecuzione entro un blocco di codice. Il profilo può essere eseguito sia sul codice utente che su quello di sistema.

- **Oggetti del dispatcher (allocatore)**: controllano l'allocazione e la sincronizzazione del sistema; di essi si parlerà nel Capitolo 6.

Windows NT non è un sistema operativo completamente orientato agli oggetti, e non è implementato in un linguaggio orientato agli oggetti. Inoltre, le strutture dati che risiedono completamente dentro un solo componente Executive non sono rappresentate come oggetti. Tuttavia, NT costituisce un buon esempio della potenza della tecnologia orientata agli oggetti, e ne testimonia la crescente diffusione nella progettazione dei sistemi operativi.

2.6 Sistemi UNIX tradizionali

Storia

La storia di UNIX è già stata narrata più volte, e non è il caso di ripercorrerla nuovamente nei dettagli: può bastare accennarvi per sommi capi. La Figura 2.14, che ne riprende una contenuta in [SALU94][2], illustra le tappe fondamentali di questa storia.

UNIX, inizialmente sviluppato nei Bell Labs, è divenuto operativo su un PDP-7 nel 1970. Alcune dei progettisti che lavoravano nei Bell Labs, avevano anche partecipato allo sviluppo del sistema time-sharing del progetto MAC al MIT, che aveva condotto allo sviluppo di CTSS e quindi di Multics. Benché di solito si affermi che UNIX è una versione ridotta di Multics, i suoi progettisti sostengono di essere stati maggiormente influenzati da CTSS [RITC78b]. Tuttavia, sono numerose le idee di Multics confluite in UNIX.

Vale la pena di accennare in breve a Multics, ricordando che, rispetto al suo tempo, era avanti non solo di anni, ma addirittura di decenni. Infatti, ancora a metà degli anni '80, quasi vent'anni dopo essere divenuto operativo, Multics possedeva caratteristiche superiori in sicurezza, nell'interfaccia utente e in altre aree, rispetto ai sistemi operativi dei mainframe dell'epoca. Benché i progettisti UNIX abbiano abbandonato il progetto Multics, credendolo destinato a fallire, Multics è passato in seguito alla Honeywell, e ha anche goduto di un modesto successo commerciale. Honeywell disponeva di altri due sistemi operativi per mainframe, uno dei quali fu lanciato sul mercato con molta determinazione, ma è difficile dire se, diversamente, Multics avrebbe potuto riscuotere un maggior successo, oppure no. In ogni caso, Multics raccolse una sua clientela, piccola ma fedele, sino alla fine degli anni '80, quando Honeywell è scomparsa dal mercato dei computer.

Nel frattempo, nei Bell Labs e poi anche altrove, venivano prodotte varie versioni di UNIX. Un primo passo fondamentale fu il trasporto di UNIX dal PDP-7 al PDP-11, il primo indizio che UNIX sarebbe diventato un sistema operativo adatto a tutti i computer. Il secondo passo, con una strategia inedita per l'epoca, fu riscrivere UNIX nel linguaggio C; generalmente, si pensava che qualcosa di complesso come un sistema operativo, che deve gestire eventi in cui il fattore tempo è critico, dovesse essere scritto esclusivamente in linguaggio assembly. L'implementazione

[2] Un albero genealogico più completo è contenuto in [MCKU96].

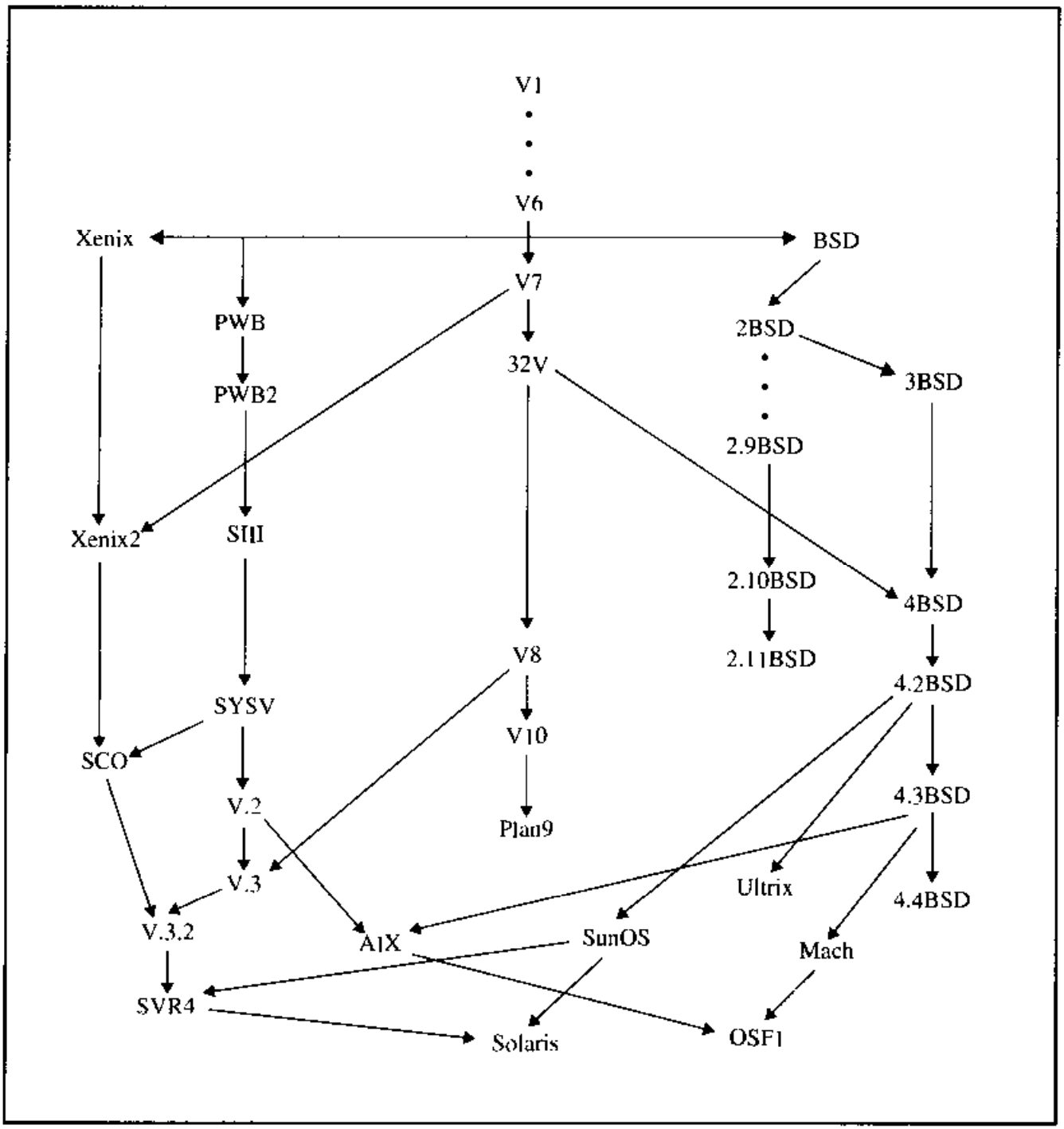


Figura 2.14 *Storia di UNIX*

in C ha invece dimostrato quanto fosse vantaggioso utilizzare un linguaggio di alto livello per scrivere, se non tutto, almeno la gran parte del codice di sistema, e, infatti, oggi quasi tutte le implementazioni di UNIX sono scritte in C.

Queste prime versioni erano abbastanza diffuse nei Bell Labs, quindi, nel 1974, il sistema UNIX venne descritto per la prima volta in una rivista specializzata [RITC74], suscitando un grande interesse. Vennero fornite licenze UNIX sia ad aziende commerciali che alle Università, e la versione 6 divenne la prima disponibile fuori dai Bell Labs, nel 1976. La versione 7 successiva, uscita nel 1978, è l'antenata della maggior parte dei sistemi UNIX di oggi. Il più importan-

te sistema non-AT&T, UNIX BSD, è stato sviluppato all'università di Berkeley, in California, ed era eseguito sui PDP, e successivamente su macchine VAX. AT&T ha continuato a sviluppare e raffinare il sistema, e dal 1982 i Bell Labs hanno combinato diverse varianti AT&T di UNIX in un singolo sistema, immesso sul mercato col nome di UNIX System III. Successivamente, per produrre UNIX System V, sono state aggiunte al sistema operativo diverse caratteristiche.

Descrizione

La Figura 2.15 presenta una descrizione generale dell'architettura di UNIX. L'hardware sottostante è circondato dal software del sistema operativo, che viene spesso chiamato kernel del sistema, o soltanto kernel, per sottolinearne la separazione dall'utente e dalle applicazioni. Quando nel testo utilizzeremo come esempio questo sistema operativo, faremo riferimento a questa porzione di UNIX. Il sistema UNIX è dotato di un insieme di servizi e interfacce per l'utente, considerate parte di esso, che si possono raggruppare nella shell, in altro software di interfaccia e nei componenti del compilatore C (compilatore, assembler, loader). Lo strato esterno a questi contiene le applicazioni utente e l'interfaccia utente del compilatore C.

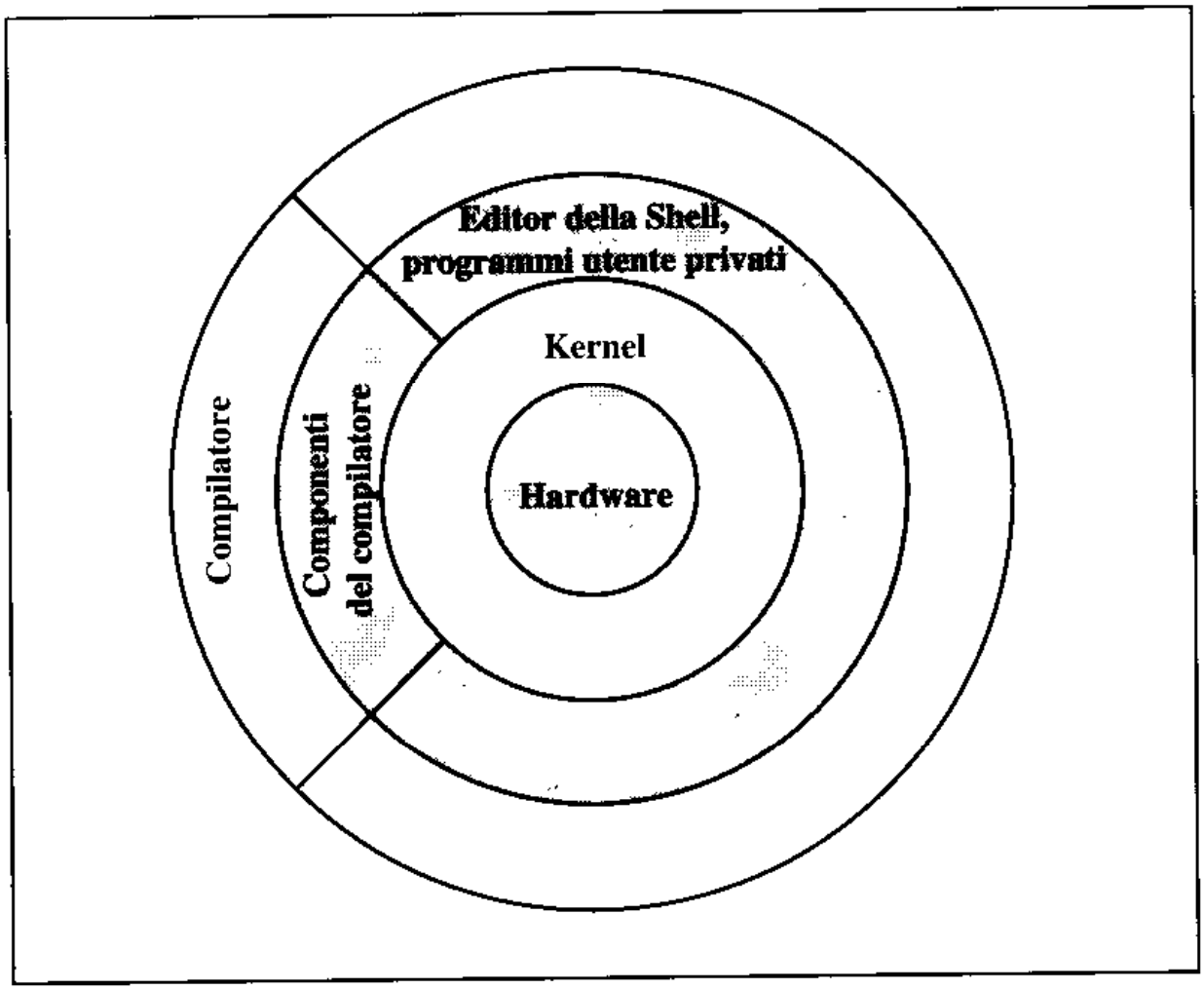


Figura 2.15 *Architettura generale di UNIX*

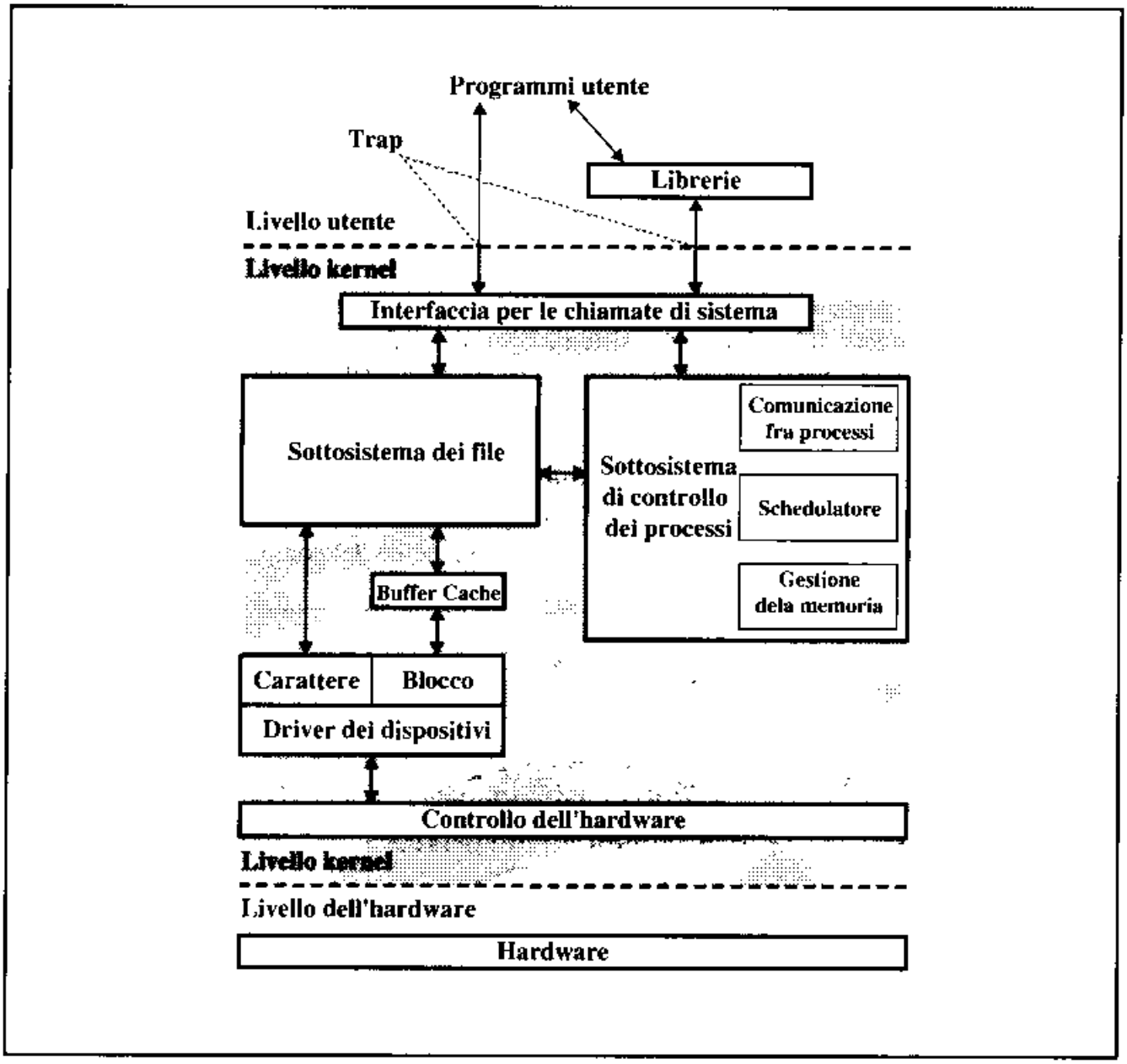


Figura 2.16 *Kernel UNIX tradizionale [BACH86]*

La Figura 2.16 consente di dare uno sguardo più ravvicinato al kernel. I programmi utente possono chiamare i servizi del sistema operativo direttamente o attraverso programmi di libreria; l'interfaccia per le chiamate di sistema rappresenta il confine tra kernel ed utente, e permette al software di più alto livello di ottenere l'accesso a funzioni specifiche del kernel; all'altro estremo, il sistema operativo contiene routine primitive che interagiscono direttamente con l'hardware. Fra queste due interfacce, il sistema è diviso in due parti principali, una delegata al controllo dei processi, l'altra alla gestione dei file e dell'I/O. Il sottosistema per il controllo dei processi è responsabile della gestione della memoria, della schedulazione e dell'allocazione dei processi, nonché della sincronizzazione e della comunicazione fra di essi. Il file system scambia i dati fra la memoria e i dispositivi esterni o attraverso flussi di caratteri o mediante blocchi, utilizzando diversi driver per i dispositivi. Per il trasferimento orientato ai blocchi si utilizza una

cache del disco: un buffer di sistema nella memoria principale è interposto fra lo spazio di indirizzamento dell'utente e il dispositivo esterno.

In questa sottosezione è stato descritto quello che si potrebbe definire un sistema UNIX tradizionale; [VAHA96] usa questo termine per riferirsi a SystemV Release3 (SVR3), 4.3 BSD e alle versioni precedenti. A proposito di questi sistemi UNIX, è possibile affermare che sono stati progettati per essere eseguiti su un singolo processore e che non sono in grado di proteggere le proprie strutture dati dall'accesso concorrente da parte di processori multipli. Il loro kernel non è molto versatile, dato che supporta un solo tipo di file system, di politica di schedulazione dei processi e di formato di file eseguibile. Il kernel UNIX tradizionale non è progettato per essere espandibile, e possiede poche funzionalità per il riutilizzo del codice. Infatti, quando sono state aggiunte nuove caratteristiche alle varie versioni di UNIX, si è dovuta scrivere una gran quantità di nuovo codice, ottenendo così un kernel sovradimensionato e non modulare.

2.7 Sistemi UNIX moderni

Man mano che UNIX si è evoluto, le sue implementazioni sono diventate numerose e ciascuna ha aggiunto alcune caratteristiche utili. Così, si è fatta strada l'esigenza di produrre una nuova implementazione che riunisse molte innovazioni importanti e che aggiungesse altre caratteristiche architetturali dei sistemi operativi moderni, rendendo la sua architettura più modulare. La Figura 2.17 illustra l'architettura tipica dei moderni kernel UNIX: un piccolo nucleo di facility, scritte modularmente, fornisce funzioni e servizi necessari a diversi processi del sistema operativo; ciascuno dei cerchi esterni rappresenta funzioni e un'interfaccia, che può essere implementata in modi diversi.

Prendiamo ora in esame alcuni sistemi UNIX moderni.

System V Release 4 (SVR4)

SVR4, sviluppato congiuntamente da AT&T e Sun Microsystems, riunisce caratteristiche di SVR3, 4.3BSD, Microsoft Xenix SystemV e SunOS, ed è una riscrittura quasi totale del kernel di SystemV, con un'implementazione pulita, anche se complessa. Questa nuova versione comprende il supporto per l'elaborazione in tempo reale, classi di schedulazione dei processi, strutture dati allocate dinamicamente, gestione della memoria virtuale, file system virtuale e kernel prerilasciabile.

SVR4 è il risultato degli sforzi dei progettisti sia commerciali sia universitari, ed è stato sviluppato per fornire una piattaforma uniforme per la diffusione commerciale di UNIX, un obiettivo che ha centrato in pieno, essendo la più importante variante di UNIX ancora in uso. SVR4 ingloba le più notevoli caratteristiche tipiche dei sistemi UNIX, integrandole in modo adatto agli usi commerciali; lo si può eseguire su macchine che variano dai microprocessori a 32 bit fino ai supercomputer ed è chiaramente uno dei più importanti sistemi operativi mai sviluppati. Molti esempi UNIX di questo libro sono tratti da SVR4.

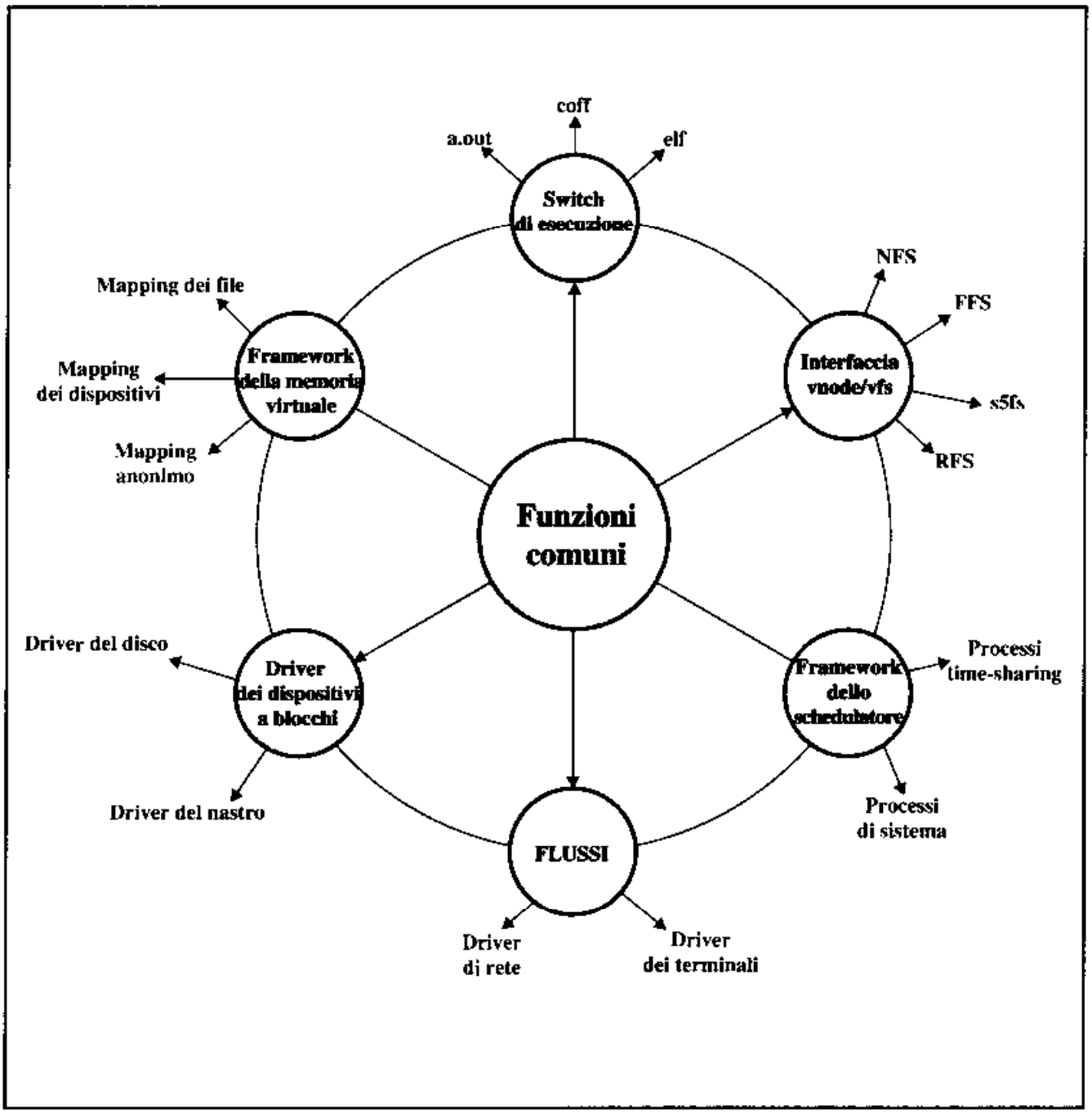


Figura 2.17 *Kernel UNIX moderno [VAHA96]*

Solaris 2.x

Solaris è la versione UNIX della Sun basata su SVR4, giunta fino alla versione 2.5. Le implementazioni della versione 2 di Solaris offrono tutte le caratteristiche di SVR4, insieme ad altre più avanzate, come un kernel pienamente prerilasciabile e multithread, il pieno supporto per SMP e un'interfaccia orientata agli oggetti per i file system. Solaris è l'implementazione UNIX più usata e di maggior successo commerciale. Per alcune caratteristiche del sistema operativo, questo libro trae qualche esempio anche da Solaris.

4.4BSD

Le versioni UNIX BSD (Berkeley Software Distribution) hanno giocato un ruolo chiave nello sviluppo teorico della progettazione dei sistemi operativi: 4.xBSD venne ampiamente utilizzato nelle Università ed è servito come base per diversi prodotti UNIX commerciali. Probabilmente, BSD è responsabile di molta della popolarità di UNIX, e i miglioramenti più significativi del sistema sono apparsi per la prima volta nelle versioni BSD.

L'ultima versione comparsa di BSD è la 4.4, quindi il gruppo che l'aveva progettata e implementata si sciolse. Si tratta di un aggiornamento considerevole di 4.3BSD, e comprende un nuovo sistema di memoria virtuale, cambiamenti nella struttura del kernel e numerosi altri miglioramenti.

2.8 I prossimi capitoli: Sommario

I primi due capitoli, che fungono da introduzione, hanno tracciato una panoramica sull'architettura del computer e sui sistemi operativi; questa sezione contiene invece un breve sommario del contenuto dei prossimi capitoli.

Descrizione e controllo dei processi

Il concetto di processo, centrale nello studio dei sistemi operativi, viene esaminato in dettaglio nel Capitolo 3, che si occupa anche di descrittori di risorse logiche e descrittori di processi. Sull'analisi degli elementi tipici di questi descrittori si basa la trattazione delle funzioni, relative ai processi, effettuate dal sistema operativo. Il capitolo descrive le primitive di controllo dei processi e presenta una descrizione degli stati (ready, running, blocked ecc.) dei processi.

Thread, SMP e microkernel

Il Capitolo 4 continua la discussione sulla gestione dei processi, esaminando caratteristiche importanti dei sistemi operativi di oggi. Un sistema operativo multithread supporta thread di esecuzione multipli entro un singolo processo, e ciò può avere effetti importanti sulle prestazioni e sulle capacità di sfruttare la concorrenza. Il multiprocessore simmetrico (SMP) è una forma di macchina a processori multipli in cui ogni processore può eseguire un'applicazione o del codice di sistema. SMP è spesso usato insieme con sistemi multithread, ma anche senza di essi può realizzare miglioramenti notevoli delle prestazioni. Infine, il Capitolo 4 si occupa anche del microkernel, che si può trovare in diversi sistemi operativi contemporanei e che cerca di minimizzare la quantità di codice del kernel del sistema operativo.

Concorrenza

I due punti qualificanti dei sistemi operativi moderni sono la multiprogrammazione e l'elaborazione distribuita. La concorrenza è fondamentale per entrambe, e quindi lo è per la proget-

tazione dei sistemi operativi. Quando molti processi sono in esecuzione concorrentemente, sorge il problema della risoluzione dei conflitti e della cooperazione: sia concretamente, nel caso di un sistema multiprocessore, sia virtualmente, nel caso di un sistema multiprogrammato a singolo processore. Il Capitolo 5 esamina i meccanismi di risoluzione dei conflitti nel contesto delle sezioni critiche dei processi che devono essere controllati. I semafori e i messaggi sono due tecniche essenziali per il controllo delle sezioni critiche, in quanto applicano il principio della mutua esclusione. Il Capitolo 6 si sofferma perciò su due problemi da non tralasciare quando si cerca di supportare l'elaborazione concorrente: lo stallo (*deadlock*) e la starvation.

Gestione della memoria

I Capitoli 7 e 8 trattano la gestione della memoria principale. Il Capitolo 7 descrive gli obiettivi della gestione della memoria relativi al problema dell'overlay e alle necessità della protezione e della condivisione. Quindi, si occupa del caricamento dei programmi e del concetto di rilocazione, per trattare poi della paginazione e della segmentazione, e dei meccanismi di indirizzamento che le supportano. Il Capitolo 8 affronta l'uso della paginazione, o della paginazione con segmentazione, per implementare il meccanismo della memoria virtuale. L'interazione fra hardware e software fa da filo conduttore in molti argomenti, dalla memoria secondaria e principale, alla cache, dalla segmentazione alla paginazione, con l'obiettivo di mostrare come sia possibile integrare tutti questi oggetti e meccanismi in uno schema complessivo di gestione della memoria.

Schedulazione

Il Capitolo 9 esordisce esaminando i tre tipi di schedulazione del processore, a lungo, a medio e a breve termine, per poi soffermarsi sul problema della schedulazione a breve termine, esaminandone e confrontandone i diversi algoritmi. Il Capitolo 10 tratta i problemi di schedulazione specifici per le configurazioni a multiprocessore, per illustrare poi come si progetti la schedulazione per sistemi in tempo reale.

Gestione dell'I/O e schedulazione del disco

Il Capitolo 11 sintetizza gli aspetti dell'I/O all'interno dell'architettura dei computer e prosegue analizzando le richieste poste dall'I/O al sistema operativo. Vengono esaminate e confrontate diverse strategie di bufferizzazione, per poi soffermarsi sui problemi relativi all'I/O del disco, fra cui la schedulazione del disco e l'uso di una cache del disco.

Gestione dei file

Il Capitolo 12 discute l'organizzazione fisica e logica dei dati; esamina i servizi relativi alla gestione dei file che un sistema operativo tipico fornisce agli utenti, e i meccanismi specifici e le strutture dati che sono parte di un sistema di gestione dei file.

Elaborazione di rete e distribuita

Sempre più diffusamente le funzionalità dei computer non si utilizzano più attraverso una singola macchina, ma come parte di una rete di computer e terminali. La prima parte del Capitolo 13 è dedicata al concetto di architettura di comunicazione, in particolare a TCP/IP. È poi la volta dell'elaborazione client/server, sempre più rilevante, e delle problematiche che essa pone ad un sistema operativo. Il resto del capitolo esamina due tecniche di comunicazione tra i processi, fondamentali per l'elaborazione distribuita: la trasmissione di messaggi distribuita e le chiamate di procedura remote. Inoltre, il Capitolo 13 si occupa del concetto di cluster, mentre nel Capitolo 14 trovano posto gli elementi chiave dei sistemi operativi distribuiti, fra cui la migrazione dei processi, la determinazione degli stati globali distribuiti, i meccanismi per la mutua esclusione, l'individuazione e la prevenzione dello stallo.

Sicurezza

Il Capitolo 15 si apre passando in rassegna i tipi di rischio cui vanno soggette le comunicazioni fra computer, e prosegue esaminando gli strumenti specifici che si possono usare per incrementare la sicurezza: per primi, gli approcci tradizionali alla sicurezza, basati sulla protezione delle diverse risorse del computer, ad esempio i dati e la memoria. Di seguito, si affronta un genere di rischio recente e sempre più preoccupante, quello causato da virus o da simili attacchi. Dopo aver esaminato un approccio alla sicurezza relativamente nuovo, cioè i sistemi fidati, il capitolo si chiude con uno sguardo alla sicurezza in rete e con un'Appendice introduttiva alla crittografia, uno strumento di base utilizzato in diverse applicazioni per la sicurezza.

Analisi delle code

L'analisi delle code è uno strumento importante, che chiunque abbia a che fare con l'informatica dovrebbe saper utilizzare. Molti problemi nel campo dei sistemi operativi e dell'elaborazione distribuita, ma anche in altri ambiti informatici, possono essere rappresentati attraverso modelli di code. L'analisi delle code permette all'analista di sviluppare rapidamente una caratterizzazione approssimata del comportamento di un sistema sottoposto a diverse tipologie di carico. L'Appendice A fornisce una panoramica dei metodi pratici per l'analisi delle code.

Progettazione orientata agli oggetti

I concetti orientati agli oggetti stanno diventando sempre più importanti nella progettazione dei sistemi operativi: ad esempio Windows NT, uno dei sistemi usati come esempio in questo libro, usa in modo sistematico queste tecniche. L'Appendice B sintetizza i principi essenziali della progettazione orientata agli oggetti.

Progetti di programmazione e sistemi operativi

A complemento di questo testo, è possibile confrontarsi con un insieme di progetti, utili per integrare e consolidare la comprensione dei concetti inerenti i sistemi operativi.

Il docente può scegliere fra i seguenti approcci:

- **Progetti di programmazione**: il manuale del docente propone un elenco di piccoli progetti di programmazione, che gli studenti possono eventualmente sviluppare per rafforzare i concetti appresi in questo libro. I progetti coprono un'ampia gamma degli argomenti trattati, ed è possibile utilizzare qualsiasi linguaggio di programmazione.
- **BACI, Ben Ari Concurrent Interpreter**: simula l'esecuzione di processi concorrenti e supporta semafori binari, semafori a contatore e monitor. Per consolidare i concetti della programmazione concorrente, è possibile svolgere diverse esercitazioni BACI, elencate nel testo.
- **Operating System Project (OSP)**: OSP è sia un'implementazione di un sistema operativo moderno, sia un ambiente flessibile per generare progetti di implementazione, adatti ad un corso introduttivo di progettazione di sistemi operativi. OSP è affiancato da diverse esercitazioni consigliate.
- **NACHOS**: come OSP, NACHOS è un ambiente per generare progetti di implementazione per consolidare i concetti appresi, ed è anche affiancato da un gruppo di esercitazioni consigliate.
- **Letture e relazioni**: il manuale del docente contiene un elenco di articoli importanti, uno per ciascun capitolo. Se ne può assegnare la lettura, e richiedere agli studenti una breve relazione che analizzi l'articolo.

L'Appendice C discute in breve questi cinque possibili approcci. L'Appendice D introduce dettagliatamente a OSP e contiene informazioni su come procurarsi il sistema, oltre ad esercizi di programmazione, mentre l'Appendice E fornisce le medesime informazioni per BACI. NACHOS è ben documentato dal suo sito Web, di cui si può trovare l'indirizzo nell'Appendice C.

Ordine degli argomenti

Dovrebbe essere naturale, per il lettore, mettere in discussione l'ordine particolare con cui gli argomenti vengono presentati in questo libro. Ad esempio, la schedulazione (Capitoli 9 e 10), strettamente correlata alla concorrenza (Capitoli 5 e 6) e al discorso generale sui processi (Capitolo 3), dovrebbe verosimilmente comparire subito dopo questi argomenti.

La difficoltà risiede nel fatto che i diversi argomenti sono strettamente correlati. Ad esempio, nel discutere la memoria virtuale, è utile potersi riferire alle problematiche della schedulazione, in relazione con i fault di pagina. Naturalmente, è anche utile potersi riferire ad alcuni problemi di gestione della memoria quando si discute delle decisioni di schedulazione. Casi di questo genere se ne potrebbero elencare all'infinito: ad esempio, per discutere della schedulazione occorre una certa conoscenza della gestione dell'I/O, e viceversa.

La Figura 2.18 rappresenta alcune fra le più evidenti relazioni fra gli argomenti. Le linee in grassetto indicano relazioni molto forti dal punto di vista delle decisioni di progettazione ed implementazione. In base a questo diagramma, dovrebbe risultare sensato partire da un'analisi

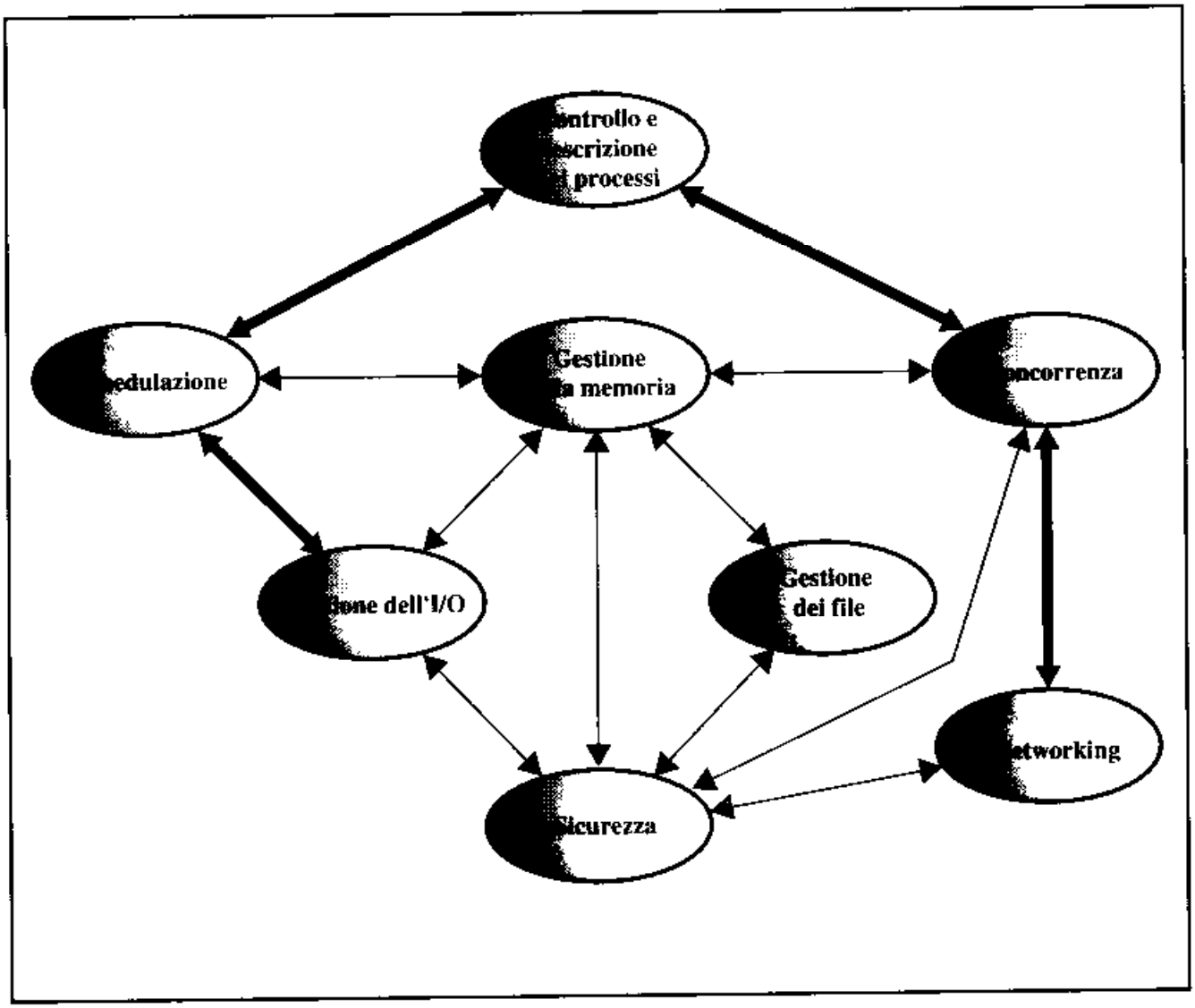


Figura 2.18 *Argomenti dei sistemi operativi*

sintetica dei processi, proposta infatti nel Capitolo 3. In seguito, l'ordine diventa per così dire arbitrario. Molti testi sui sistemi operativi radunano tutto il materiale sui processi all'inizio e poi procedono con gli altri argomenti, e si tratta di una scelta indubbiamente valida. Tuttavia, l'estrema rilevanza della gestione della memoria, che io ritengo un problema cruciale quanto la gestione dei processi, mi ha indotto a parlare di questo argomento, prima di affrontare lo studio approfondito della schedulazione.

La soluzione ideale per lo studente, dopo aver completato la lettura dei capitoli dall'1 al 3, sta nel leggere ed assimilare in parallelo i contenuti di questi capitoli: il 4 seguito (volendo) dal 5; il 6 seguito dal 7; l'8 seguito (volendo) dal 9 e il 10. Quindi, in un ordine qualunque, l'11 o il 12, seguito dal 13 o dal 14. Comunque, benché anche il cervello umano possa cimentarsi nell'elaborazione parallela, lo studente, in quanto umano, giudica impossibile (e costoso) studiare con profitto su quattro copie del medesimo libro aperte simultaneamente su 4 capitoli diversi: visto che un ordine lineare è necessario, mi pare che quello proposto sia il più conveniente.

2.9 Letture raccomandate

Esistono molti libri sui sistemi operativi. [SILB94], [TANE92] e [MILE92], che utilizzano un insieme di sistemi operativi importanti come casi di studio, esauriscono tutti i principi fondamentali. [SING94] e [NUTT92] trattano gli argomenti ad un livello più avanzato.

Passando ai testi dedicati ai sistemi che abbiamo preso ad esempio, un'eccellente trattazione della struttura interna di UNIX, completa di un'analisi comparativa di diverse varianti, è contenuta in [VAHA96]. Per UNIX SVR4, [GOOD94] fornisce una trattazione esauriente, con ampi dettagli tecnici; un testo più datato, ma ancora utile, è quello di [BACH86]. Per Berkeley UNIX 4.4 BSD, ben noto in ambiente accademico, [MCKU96] è particolarmente raccomandabile. Importanti raccolte di articoli su UNIX sono state pubblicate nel numero di luglio-agosto 1978 di *Bell System Technical Journal* e nel numero di ottobre 1984 del *AT&T Bell Laboratories Technical Journal*, poi ristampati in [ATT87a e b].Una trattazione concisa ma esauriente di Solaris 2.x è in [GRAH95].

Non molto è stato scritto sulla struttura interna di Windows NT: [CUST93] ne dà un'eccellente trattazione.

Il lettore interessato al sistema operativo VMS, eseguito sulle macchine DEC VAX, ricordi che si tratta di uno dei sistemi meglio progettati e documentati, e maggiormente studiati. Il testo fondamentale è [KENA91], che costituisce anche un modello di documentazione di un sistema operativo.

ATT87a AT&T *UNIX System readings and Examples, Volume I*. Englewood Cliffs, NJ:Prentice Hall, 1987.

ATT87b AT&T *UNIX System readings and Examples, Volume II*. Englewood Cliffs, NJ:Prentice Hall, 1987.

BACH86 Bach, M. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ:Prentice Hall, 1986. Edizione italiana *UNIX Architettura di sistema*, Jackson Libri, 1987.

CUST93 Custer, H. *Inside Windows NT*. Redmond, WA: Microsoft Press, 1993.

GOOD94 Goodheart, B., e Cox, J. *The Magic Garden Explained: The Internals Of UNIX System V Release 4*. Englewood Cliffs, NJ:Prentice Hall, 1994.

GRAH95 Graham, J. *Solaris 2.x: Internals and Architecture*. New York: Mc Graw Hill, 1995.

KENA91 Kenah, L., Goldenberg, R.; e Bare, S. *VAX/VMS Internals and Data Structures*. Bedford, MA: Digital Press, 1991.

MCKU96 McKusick, M., Bostic, K., Karels, M. e Quartermain J. *The Design and Implementation of the 4.4 BSD UNIX Operating System*. Reading, MA: Addison-Wesley, 1996.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New York: Mc Graw Hill, 1992.

NUTT92 Nutt, G. *Centralized and Distributed Operating Systems*. Englewood Cliffs, NJ:Prentice Hall, 1992.

SILB94 Silberschatz A., e Galvin P. *Operating Systems Concepts*. Reading, MA: Addison-Wesley, 1994.

SING94 Singhal, M. e Shivat\ratri, N. *Advanced Concepts in Operating Systems*. New York: Mc Graw Hill, 1994.

TANE92 Tanenbaum, A. *Modern Operating Systems*. Englewood Cliffs, NJ:Prentice Hall, 1992.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.

2.10 Problemi

2.1 Si supponga di aver un computer multiprogrammato in cui ciascun job ha le medesime caratteristiche. In un periodo di computazione T, per un job metà del tempo è passato in attività di I/O e l'altra metà in attività del processore. Ciascun job viene eseguito per un totale di N periodi. Si supponga che sia usata una semplice priorità round-robin, e che le operazioni di I/O possano sovrapporsi a quelle del processore. Si definiscano le seguenti quantità:

- Tempo di turnaround = tempo effettivo di completamento di un job.
- Throughput = numero medio di job completati per periodo di tempo T.
- Utilizzo del processore = percentuale di tempo in cui il processore è attivo (non in attesa).

Si calcolino queste quantità per 1, 2 e 4 job simultanei, supponendo che il periodo T sia distribuito nei modi seguenti:

a. per la prima metà I/O; per la seconda, processore

b. I/O per il primo ed ultimo quarto, processore per il secondo e terzo quarto

2.2 Un programma si dice I/O bound se, eseguito da solo, passerebbe più tempo in attesa di I/O che utilizzando il processore. Un programma processor-bound è l'opposto. Si supponga che un algoritmo di schedulazione a breve termine favorisca quei programmi che hanno usato poco tempo di processore nel recente passato. Si spieghi perché questo algoritmo favorisca i programmi I/O-bound, e tuttavia non neghi in eterno il tempo di processore ai programmi processor-bound.

2.3 Un computer ha una cache, una memoria principale e un disco utilizzato per la memoria virtuale. Se una parola è nella cache sono richiesti A ns per accedervi. Se è nella memoria centrale ma non nella cache, sono necessari B ns per caricarla nella cache, quindi viene nuovamente riavviato il riferimento alla memoria. Se la parola non è nella memoria principale, sono necessari C ns per prelevarla dal disco, seguiti da B ns per metterla nella cache. Se la hit ratio della cache è $(n-1)/n$, e la hit ratio della memoria principale è $(m-1)/m$, qual è il tempo di accesso medio?

2.4 Confrontare le politiche di schedulazione che si dovrebbero usare quando si ottimizza un sistema time-sharing, con quelle che si dovrebbero utilizzare per ottimizzare un sistema batch multiprogrammato.

2.5 Qual è la finalità delle chiamate di sistema e come fanno a mettersi in relazione con il sistema operativo e con il concetto di operazione dual-mode?

2.6 Nel sistema operativo per mainframe dell'IBM, l'OS/390, uno dei principali moduli del kernel è il gestore delle risorse di sistema (SRM, System Resource Manager). Questo modulo è responsabile dell'allocazione delle risorse fra gli spazi di indirizzamento (processi). SRM fornisce a OS/390 un grado di sofisticazione unico fra i sistemi operativi: nessun altro sistema per mainframe, e certo nessun altro sistema in generale, può uguagliare le funzioni effettuate da SRM. Il concetto di risorsa comprende il processore, la memoria reale e i canali di I/O. SRM raccoglie dati statistici relativi agli utilizzi del processore, dei canali e delle diverse strutture dati fondamentali. Il suo fine è fornire prestazioni ottimali basandosi sul monitoraggio e l'analisi delle prestazioni. L'installazione stabilisce all'inizio vari obiettivi per le prestazioni, che servono come guida per SRM, il quale modifica dinamicamente le caratteristiche di prestazione dell'installazione e dei job basandosi sull'utilizzo del sistema. SRM a sua volta fornisce informazioni che permettono all'operatore esperto di mettere a punto la configurazione e i valori dei parametri, per migliorare il servizio verso gli utenti. Questo problema riguarda un esempio di attività SRM. La memoria reale è divisa in blocchi di uguale dimensioni chiamati frame, che possono essere migliaia. Ciascun frame può contenere un blocco di memoria virtuale chiamato pagina. SRM riceve il controllo approssimativamente 20 volte al secondo e ispeziona ciascun frame di pagina. Se la pagina non è stata referenziata o cambiata, un contatore viene incrementato di 1. Dopo un certo tempo, SRM effettua la media di questi valori per determinare il numero medio di secondi in cui un frame di pagina resta intatto. Quale potrebbe essere lo scopo di queste operazioni di SRM? Quali azioni potrebbe intraprendere SRM?

Appendice 2A Internet e risorse Web

Per integrare il testo nell'edizione originale, Internet e il Web offrono parecchie risorse e che aiutano a mantenersi aggiornati sugli sviluppi in questo campo.

Siti Web collegati al testo

Una pagina Web creata appositamente per il testo si trova all'indirizzo http://www.williamstallings.com. Il sito, totalmente sviluppato in inglese, contiene:

- Collegamenti ad altri siti Web, compresi quelli elencati qui in appendice, che fungono da ponte verso altre rilevanti risorse sul Web.
- Lucidi della maggior parte delle figure presenti nel testo, in formato PDF (Adobe Acrobat).
- Una serie di lucidi, da utilizzare per le lezioni o come appunti da distribuire.

- Spero anche di includere link verso le home page di corsi basati su questo libro; tali pagine potrebbero essere utili ad altri insegnanti come fonte di idee sull'organizzazione di un corso.

Gli eventuali errori, di stampa o di altro genere, saranno elencati in una errata corrige disponibile nel sito che verrà di volta in volta aggiornata.

Altri siti Web

I siti Web che offrono informazioni[3] inerenti i contenuti di questo testo sono numerosi:

- Windows NT (http://www.harbornet.com/winnt/sources.html): informazioni su Windows NT, oltre a collegamenti a prodotti, interfacce utente ed informazioni tecniche.
- UNIX Reference Desk (http://www.eecs.nwu.edu/unix.html): informazioni su UNIX, oltre a collegamenti a prodotti, interfacce utente ed informazioni tecniche.
- Le FAQ (Frequently Asked Questions) di comp.os.research (http://www.best.com:80/~bos/os-faq): ampie ed utili FAQ che coprono problematiche di progetto dei sistemi operativi.
- ACM Special Interest Group on Operating Systems (Http://www.acm.or/sigops): informazioni sulle pubblicazioni e le conferenze di SIGOPS.

Newsgroup di USENET

Diversi newsgroup di USENET sono dedicati ad alcuni aspetti dei sistemi operativi o ad un particolare sistema operativo. Praticamente tutti i newsgroup USENET soffrono di un elevato rapporto segnale/rumore, ma si può provare a vedere se qualcuno può risolvere i nostri problemi. Questi sono i due newsgroup più importanti:

- comp.os.research: è il più interessante: si tratta di un newsgroup con moderatore che affronta argomenti di ricerca.
- comp.os.misc: un newsgroup per discussioni generali sugli argomenti relativi ai sistemi operativi.

[3] Si tenga presente che il materiale su Web potrebbe cessare di essere raggiungibile senza preavviso; la lista che segue e gli altri riferimenti del testo si riferiscono a Febbraio 2000.

PROCESSI

Parte 2

Il compito fondamentale di un moderno sistema operativo consiste nella gestione dei processi; è suo compito allocare risorse ai processi, permettere loro di condividere e scambiare informazioni, proteggere le risorse di ciascun processo dagli altri, permetterne la sincronizzazione. Per soddisfare queste necessità, il sistema operativo deve mantenere per ciascun processo una struttura dati, che descrive lo stato e la proprietà delle risorse dei processi, e che gli permette di esercitare il controllo dei processi stessi.

Nella multiprogrammazione su un singolo processore, l'esecuzione di processi può avvenire in modo interallacciato nel tempo, mentre nei sistemi multiprocessore essa può essere non solo interallacciata, ma anche simultanea. Interallacciamento ed esecuzione simultanea sono due forme di concorrenza che generano diversi problemi complessi, sia per il programmatore delle applicazioni, sia per il sistema operativo.

In molti sistemi operativi d'oggi, le difficoltà di gestione dei processi sono accentuate dall'introduzione del concetto di thread. In un sistema multithread il processo conserva gli attributi di proprietà delle risorse, mentre gli attributi dei flussi d'esecuzione, multipli e concorrenti, sono di proprietà dei thread in esecuzione all'interno di un processo.

Capitolo 3
Descrizione e controllo dei processi

Il punto focale di un sistema operativo tradizionale è la gestione dei processi. Ciascun processo è in ogni momento in un certo stato di esecuzione, come Ready, Running, Blocked, di cui il sistema operativo conserva traccia, gestendo i cambiamenti di stato. Per questo, il sistema operativo mantiene strutture dati molto complesse, che descrivono ciascun processo; inoltre effettua la schedulazione e fornisce facilitazioni per la condivisione e sincronizzazione dei processi. Il Capitolo 3 esami-

na le strutture dati e le tecniche usate da un tipico sistema operativo per gestire i processi.

Capitolo 4
Thread, SMP e microkernel

Il Capitolo 4 si occupa di tre aree, caratteristiche di molti sistemi operativi contemporanei, che costituiscono un progresso rispetto all'architettura di quelli tradizionali. In molti di tali sistemi, il tradizionale concetto di processo appare scisso in due parti: una relativa alle proprietà delle risorse (il processo), l'altra al flusso di esecuzione delle istruzioni (il thread). Un singolo processo può contenere thread multipli, e un'organizzazione multithread è vantaggiosa sia nella strutturazione delle applicazioni sia per le prestazioni. Il capitolo esamina anche il multiprocessore simmetrico (SMP), un sistema con processori multipli ciascuno dei quali può eseguire applicazioni oppure codice di sistema, incrementando prestazioni e affidabilità. L'ultima parte si occupa del microkernel, un modo di progettare il sistema operativo che minimizza la quantità di codice eseguita in modalità kernel, e ne analizza i vantaggi.

Capitolo 5
Concorrenza: mutua esclusione e sincronizzazione

Nella progettazione dei sistemi operativi, la concorrenza costituisce un problema complesso: il capitolo ne esamina due aspetti, la sincronizzazione e la mutua esclusione. Quest'ultima è in rapporto alla capacità dei processi multipli (o thread) di condividere codice, risorse o dati, in modo che un solo processo per volta acceda all'oggetto condiviso. La sincronizzazione, correlata alla mutua esclusione, si può definire come la capacità dei processi multipli di coordinare le proprie funzioni attraverso lo scambio di informazioni. Il capitolo contiene un minuzioso esame delle problematiche inerenti la concorrenza, a partire dalle difficoltà in sede di progettazione, quindi si occupa del supporto hardware alla concorrenza e dei principali meccanismi che la supportano: semafori, monitor, scambi di messaggi.

Capitolo 6
Concorrenza: stallo e starvation

Il Capitolo 6 approfondisce altri due aspetti della concorrenza. Parliamo di stallo quando due o più processi si attendono reciprocamente per completare un'operazione, ma nessuno di essi è in grado di proseguire. Prevedere uno stallo è difficile e non esistono facili soluzioni generali

a tale problema. Il capitolo passa in rassegna i tre approcci principali al problema dello stallo: prevenirlo, evitarlo e rilevarlo. Con starvation (*morte per fame*) ci si riferisce alla situazione in cui un processo è pronto per l'esecuzione, ma gli è continuamente negato l'accesso al processore, a favore degli altri. In larga parta la starvation è trattata come un problema di schedulazione e perciò se ne discute nella Parte Quarta; benché la condizione di stallo sia il tema centrale di questo capitolo, si affronta il problema della starvation nell'ambito delle soluzioni allo stallo che consentono di evitarla.

CAPITOLO 3

DESCRIZIONE E CONTROLLO DEI PROCESSI

La progettazione di un sistema operativo deve rispondere ai bisogni per soddisfare i quali è stata intrapresa. Tutti i sistemi multiprogrammati, da quelli a singolo utente come Windows 95, a sistemi per mainframe come OS/390, che possono supportare migliaia di utenti, sono costruiti attorno al concetto di processo, al quale ci si può riferire elencando la maggior parte delle richieste che un sistema operativo deve soddisfare:

- Interallacciare l'esecuzione di diversi processi, per massimizzare l'uso del processore; contemporaneamente, fornire un tempo di risposta ragionevole.
- Allocare risorse ai processi, in conformità ad una specifica politica (ad esempio, dando maggiore priorità a determinate funzioni o applicazioni), e nello stesso tempo evitare lo stallo[1].
- Supportare la comunicazione fra i processi e la loro creazione a livello utente, entrambe utili per strutturare le applicazioni.

Iniziamo lo studio dettagliato dei sistemi operativi partendo da come essi rappresentino e controllino i processi, esaminando gli stati che caratterizzano il comportamento dei processi. Passiamo quindi alle strutture dati necessarie al sistema per rappresentare lo stato di ciascun processo e alle caratteristiche di cui il sistema operativo ha bisogno per raggiungere i suoi obiettivi, per concludere con la gestione dei processi in UNIX SVR4.

---

[1] Il problema dello stallo sarà esaminato nel Capitolo 6. In pratica, avviene uno stallo quando due processi hanno bisogno delle stesse due risorse per continuare, ciascuno di essi ne possiede una, e aspetta indefinitamente quella di cui non dispone.

3.1 Stati dei processi

La funzione principale di un processore consiste nell'eseguire istruzioni macchina, che risiedono nella memoria principale come programmi. Com'è noto dal precedente capitolo, un processore può interallacciare l'esecuzione dei programmi.

Dal punto di vista del processore, le istruzioni sono eseguite in una sequenza, determinata dai cambiamenti dei valori nel registro contatore di programma; nel tempo, il contatore di programma può riferirsi al codice di diversi programmi, appartenenti a diverse applicazioni. Dal punto di vista di un programma, l'esecuzione comporta una sequenza di istruzioni entro il programma stesso, e viene definita come processo o task.

È possibile caratterizzare il comportamento di un singolo processo, elencando la relativa sequenza di istruzioni eseguite; tale elenco è definito *traccia* del processo ([HOAR85] contiene una rigorosa trattazione in proposito). Il comportamento del processore si può invece caratterizzare mostrando come siano interallacciate le tracce dei diversi processi.

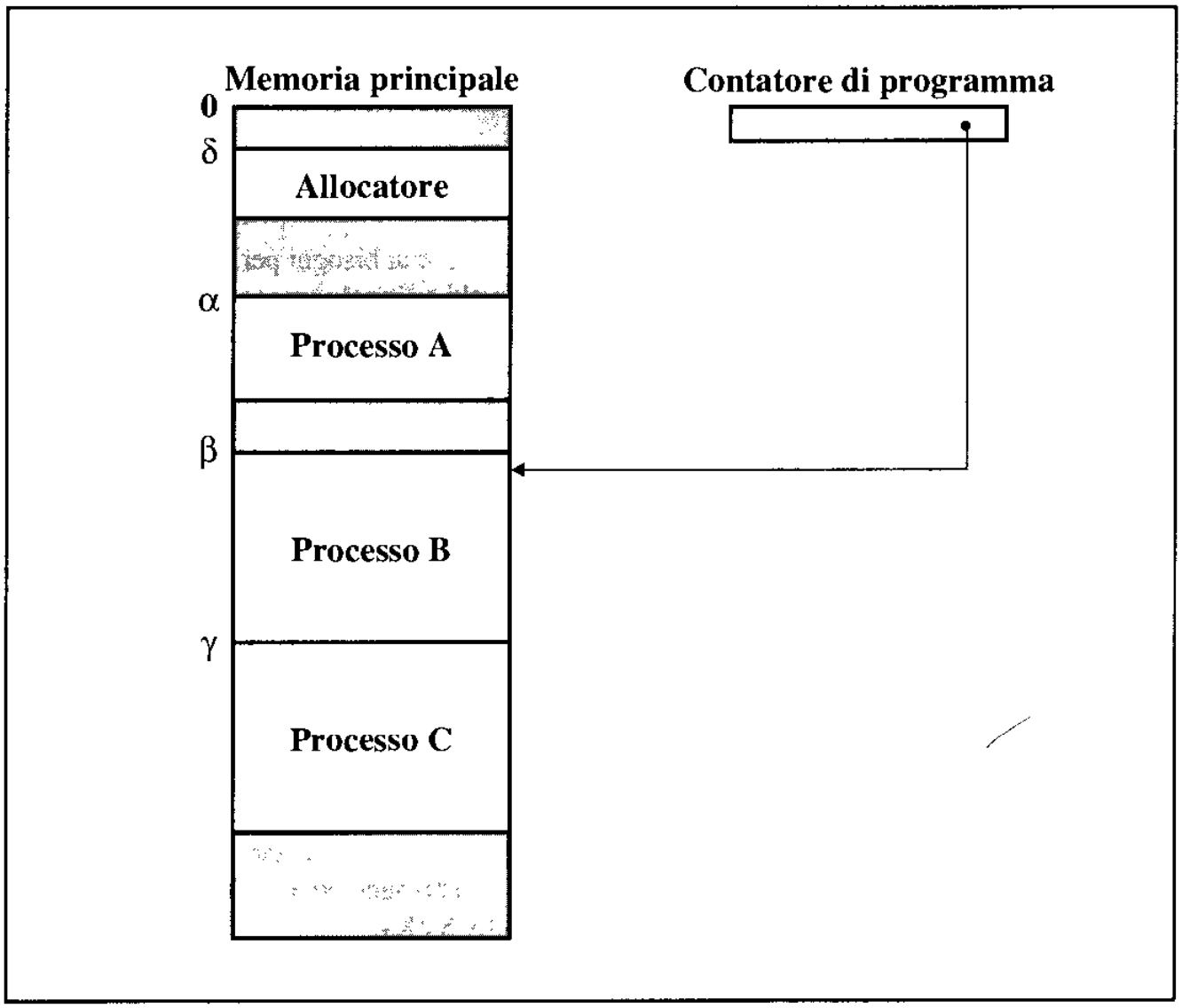


Figura 3.1 *Istantanea dell'esecuzione di esempio (Figura 3.3) al ciclo di istruzione 16*

Esaminiamo un caso molto semplice, la disposizione in memoria di tre processi della Figura 3.1, supponendo inoltre di non far uso della memoria virtuale; tutti e tre i processi sono rappresentati da programmi totalmente caricati nella memoria principale. Esiste poi un piccolo programma allocatore, che trasferisce il processore da un processo ad un altro. La Figura 3.2 mostra le tracce dei tre processi nella prima parte della loro esecuzione, con le prime 12 istruzioni eseguite nei processi A e C. Il processo B esegue 4 istruzioni, l'ultima delle quali chiama un'operazione di I/O, perciò il processo deve aspettare.

Consideriamo ora le stesse tracce dal punto di vista del processore; la Figura 3.3 mostra le tracce interallacciate che sono dai primi 52 cicli di istruzione (per comodità, i cicli sono numerati). Il sistema operativo, nella nostra ipotesi, permette ad un processo di continuare l'esecuzione per un massimo di 6 cicli di istruzione, dopodiché lo interrompe, impedendogli così di monopolizzare il tempo di processore. Come si vede in figura, sono eseguite le prime 6 istruzioni del processo A, seguite da un'interruzione e dall'esecuzione di codice dell'allocatore, che passa il controllo al processo B[2]. Eseguite quattro istruzioni, il processo B richiede un'operazione di I/O, perciò deve attendere: il processore cessa di eseguirlo e si sposta, attraverso l'allocatore, ad eseguire il processo C, per poi ritornare ad A, al termine del quanto di tempo di C. Allo scadere del quanto di tempo di A, poiché il processo B sta ancora aspettando il completamento dell'operazione di I/O, l'allocatore assegna l'esecuzione nuovamente al processo C.

$\alpha + 0$	$\beta + 0$	$\gamma + 0$
$\alpha + 1$	$\beta + 1$	$\gamma + 1$
$\alpha + 2$	$\beta + 2$	$\gamma + 2$
$\alpha + 3$	$\beta + 3$	$\gamma + 3$
$\alpha + 4$		$\gamma + 4$
$\alpha + 5$		$\gamma + 5$
$\alpha + 6$		$\gamma + 6$
$\alpha + 7$		$\gamma + 7$
$\alpha + 8$		$\gamma + 8$
$\alpha + 9$		$\gamma + 9$
$\alpha + 10$		$\gamma + 10$
$\alpha + 11$		$\gamma + 11$
(a) Traccia del processo A	**(b) Traccia del processo B**	**(c) Traccia del processo C**

α = Indirizzo di inizio del programma del processo A
β = Indirizzo di inizio del programma del processo B
γ = Indirizzo di inizio del programma del processo C

Figura 3.2 *Tracce dei processi della Figura 3.1*

[2] Il numero di istruzioni eseguite per i processi e l'allocatore in realtà è sempre molto più alto; l'esempio usa numeri piccoli soltanto per chiarire il discorso.

1	$\alpha+0$	
2	$\alpha+1$	
3	$\alpha+2$	
4	$\alpha+3$	
5	$\alpha+4$	
6	$\alpha+5$	Tempo di esecuzione scaduto
7	$\delta+0$	
8	$\delta+1$	
9	$\delta+2$	
10	$\delta+3$	
11	$\delta+4$	
12	$\delta+5$	
13	$\beta+0$	
14	$\beta+1$	
15	$\beta+2$	
16	$\beta+3$	
		Richiesta di I/O
17	$\delta+0$	
18	$\delta+1$	
19	$\delta+2$	
20	$\delta+3$	
21	$\delta+4$	
22	$\delta+5$	
23	$\gamma+0$	
24	$\gamma+1$	
25	$\gamma+2$	
26	$\gamma+3$	
27	$\gamma+4$	
28	$\gamma+5$	Tempo di esecuzione scaduto
29	$\delta+0$	
30	$\delta+1$	
31	$\delta+2$	
32	$\delta+3$	
33	$\delta+4$	
34	$\delta+5$	
35	$\alpha+6$	
36	$\alpha+7$	
37	$\alpha+8$	
38	$\alpha+9$	
39	$\alpha+10$	
40	$\alpha+11$	Tempo di esecuzione scaduto
41	$\delta+0$	
42	$\delta+1$	
43	$\delta+2$	
44	$\delta+3$	
45	$\delta+4$	
46	$\delta+5$	
47	$\gamma+6$	
48	$\gamma+7$	
49	$\gamma+8$	
50	$\gamma+9$	
51	$\gamma+10$	
52	$\gamma+11$	Tempo di esecuzione scaduto

δ = Indirizzo di inizio del programma allocatore

Figura 3.3 *Tracce combinate dei processi della Figura 3.1*

Un modello di processo a due stati

Compito principale del sistema operativo è il controllo dell'esecuzione dei processi, esercitato anche determinando lo schema di interallacciamento per l'esecuzione e l'allocazione delle risorse. Quando si progetta un programma per il controllo dei processi, il primo passo consiste nel descrivere il comportamento che ci si aspetta di riscontrare in essi.

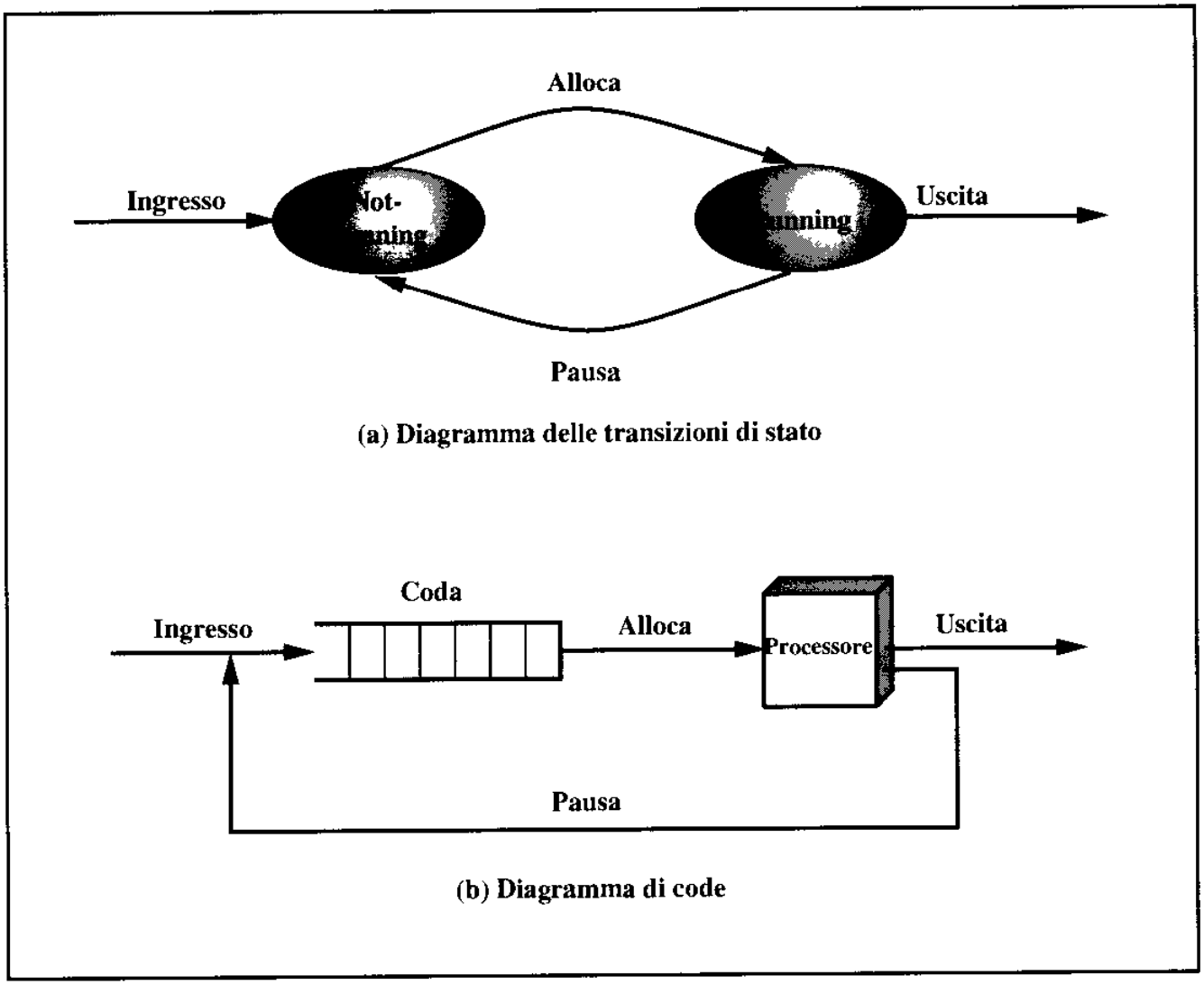


Figura 3.4 *Modello di processo a due stati*

Possiamo costruire un modello semplicissimo, osservando che in ogni momento un processo è (oppure non è) eseguito da un processore, cioè si trova in uno di questi due stati: Running o Not-Running (Figura 3.4a). Quando il sistema operativo crea un nuovo processo, lo introduce nel sistema nello stato di Not-Running: il processo esiste, è noto al sistema operativo e aspetta l'occasione di essere eseguito: da un momento all'altro, il processo correntemente in esecuzione sarà interrotto, e la porzione allocatore del sistema operativo selezionerà un nuovo processo per l'esecuzione; il processo corrente passa così dallo stato di Running a quello di Not-Running, mentre uno dei processi in attesa passa nello stato di Running.

Questo modello, per quanto semplice, ci consente di apprezzare alcuni degli elementi della progettazione di un sistema operativo: ogni processo deve essere rappresentato in modo che il sistema operativo possa conservarne traccia, in altre parole devono esistere informazioni relative a ciascun processo, incluso lo stato corrente e la locazione in memoria. I processi in attesa di essere eseguiti devono essere tenuti in qualche tipo di coda, di cui la Figura 3.4b suggerisce una possibile struttura: una sola coda, dove ciascun elemento è un puntatore ad un particolare processo; in alternativa, la coda può consistere in una lista di blocchi di dati, in cui ciascun blocco rappresenta un processo (vedremo più avanti questa seconda implementazione).

Si può descrivere il comportamento dell'allocatore con questo diagramma a code: un processo interrotto è trasferito alla coda dei processi in attesa, oppure, se il processo ha terminato oppure ha fallito, è scaricato, in altre parole esce dal sistema. In entrambi i casi, l'allocatore passa a selezionare, dalla coda, un nuovo processo per l'esecuzione.

Creazione e terminazione dei processi

Prima di aumentare la complessità del nostro semplice modello a due stati, sarà utile esaminare la creazione e la terminazione dei processi, in altre parole gli estremi fra cui è compresa la vita di un processo, a prescindere dal suo modello di comportamento.

Creazione dei processi

Quando è aggiunto un nuovo processo, il sistema operativo costruisce le strutture dati utili a gestirlo (vedi in proposito la Sezione 3.2) e alloca lo spazio d'indirizzamento: creare un nuovo processo, infatti, significa proprio compiere queste due azioni.

La creazione di un processo, di norma è determinata da quattro eventi, come indicato nella Tabella 3.1: in un ambiente batch, un processo è creato in risposta alla presentazione di un job, mentre in un ambiente interattivo ciò accade quando un nuovo utente cerca di accedere al sistema. In entrambi i casi, il sistema operativo è responsabile della creazione dei nuovi processi. Un sistema operativo può anche creare un processo per conto di un'applicazione: ad esempio, se un utente richiede la stampa di un file, il sistema crea un processo che gestirà la stampa, ed il processo che ha effettuato la richiesta può quindi proseguire, indipendentemente dal tempo richiesto per completare il task di stampa.

In passato, i sistemi operativi creavano tutti i processi in modo trasparente per l'utente o per il programma applicativo, come peraltro accade tuttora in molti casi. Permettere che un processo ne crei direttamente un altro può risultare utile: ad esempio, un processo di un'applicazione può generarne un altro, per ricevere i dati prodotti dall'applicazione e per organizzarli in una forma utilizzabile per un'analisi successiva; il nuovo processo viene eseguito in parallelo all'ap-

Tabella 3.1 *Cause della creazione dei processi*

Nuovo job batch	Al sistema operativo arriva un flusso di controllo di job batch, solitamente da nastro o da disco. Quando il sistema operativo è pronto a ricevere un nuovo lavoro, leggerà la sequenza successiva di comandi per il controllo dei job.
Ingresso interattivo (logon) nel sistema	Un utente al terminale si collega al sistema
Creazione da parte del sistema operativo per fornire un servizio	Il sistema operativo può creare un processo per svolgere una funzione per conto di un programma utente, senza che l'utente debba aspettare (ad es. una stampa).
[illegible]	Per fini di modularità o per sfruttare il parallelismo, un programma utente può determinare la creazione di nuovi processi.

plicazione, ed è attivato ogni tanto, quando sono disponibili nuovi dati: questo modo di procedere può essere molto utile nella strutturazione delle applicazioni. Un secondo esempio potrebbe essere quello di un processo server (pensiamo ad un server per la stampa o per i file), che può generare un nuovo processo per ciascuna richiesta che gestisce. Quando il sistema operativo crea un processo (processo figlio) su esplicita richiesta di un altro (processo padre), tale azione si definisce come *generazione di processi.*

Questi processi "imparentati" di solito hanno bisogno di comunicare e cooperare, ponendo così al programmatore compiti piuttosto difficili da risolvere, come vedremo più avanti, nel Capitolo 5.

Terminazione dei processi

La Tabella 3.2, basata su [PINK89], riassume le cause più comuni di terminazione dei processi. I sistemi di elaborazione devono fornire ai processi un modo per indicare il proprio completamento. Un job batch dovrebbe contenere un'istruzione Halt, oppure una chiamata esplicita ad un servizio del sistema operativo per la terminazione; nel primo caso, l'istruzione Halt genera un'interruzione per avvertire il sistema che il processo è giunto al termine. In un'applicazione interattiva, sarà un'azione dell'utente ad indicare quando il processo è terminato. Ad esempio, in un sistema time-sharing il processo di un particolare utente deve terminare quando egli si scollega o spegne il suo terminale, mentre su di un personal computer o una workstation il processo termina quando l'utente esce da un'applicazione (ad esempio, un programma di scrittura, un foglio elettronico). Si tratta, infatti, di azioni che conducono tutte ad una richiesta di servizio al sistema operativo, per far terminare il processo richiedente.

Anche un certo numero di errori e di guasti può condurre alla terminazione di un processo; la Tabella 3.2 ne elenca alcuni dei più comuni[3].

Ricordiamo infine che, in alcuni sistemi operativi, un processo può essere fatto terminare dal processo che lo ha creato, o quando termina il processo genitore stesso.

Un modello a cinque stati

Se tutti i processi fossero sempre pronti per l'esecuzione, la strategia di accodamento suggerita dalla Figura 3.4b sarebbe sempre efficace: la coda è costituita, infatti, da una lista first-in first-out (il primo elemento ad entrare è il primo ad uscire), ed il processore opera in modo round-robin sui processi disponibili (ciascun processo dispone di una certa quantità di tempo per l'esecuzione alla cui scadenza ritorna in coda, a meno che non sia bloccato). Tuttavia, anche in un caso semplice come quello del nostro esempio, questa implementazione è inadeguata. Infatti, alcuni processi nello stato di Not-Running sono pronti per l'esecuzione, mentre altri sono bloccati in attesa del completamento di un'operazione di I/O: utilizzando una singola coda,

[3] Un sistema operativo "comprensivo" potrebbe, in alcuni casi, permettere all'utente di riprendere l'esecuzione dopo un errore, senza far terminare il processo. Ad esempio, se un utente richiede di accedere a un file e tale accesso gli è negato, il sistema operativo potrebbe semplicemente comunicarlo all'utente, e far proseguire il processo.

Tabella 3.2 *Cause della terminazione dei processi*

Terminazione normale	Il processo esegue una chiamata di servizio al sistema operativo per indicare che ha completato l'esecuzione
Superamento del tempo massimo	Il processo è stato eseguito più a lungo del tempo limite totale specificato. Esistono diverse possibilità di misurare il tempo: il tempo totale trascorso ("tempo di orologio"), il tempo totale di esecuzione e, nel caso di un processo interattivo, il tempo trascorso dall'ultimo input fornito dall'utente.
Memoria non disponibile	Il processo richiede più memoria di quella che il sistema può fornire.
Violazione dei limiti di memoria	Il processo tenta di accedere ad una locazione di memoria non consentita.
Errore di protezione	Il processo tenta di utilizzare una risorsa o un file che non gli è permesso utilizzare, o tenta di usarla in modo improprio, come scrivere su un file a sola lettura.
Errore aritmetico	Il processo tenta di eseguire un calcolo proibito, come una divisione per zero, o tenta di memorizzare numeri più grandi di quelli che l'hardware possa contenere.
Tempo scaduto	Il processo ha atteso un tempo superiore al massimo consentito perchè avvenga un determinato evento.
Fallimento di un'operazione di I/O	Avviene un errore durante un'operazione di input o output, come l'impossibilità di trovare un file, un fallimento di lettura o scrittura dopo un determinato numero massimo di tentativi (quando, per esempio, viene incontrata un'area danneggiata su un nastro), o un'operazione non valida (come una lettura dalla stampante).
Istruzione non valida	Il processo tenta di eseguire un'operazione non esistente (un salto in un'area dati ed un tentativo di eseguire i dati).
Istruzioni privilegiate	Il processo tenta di eseguire un'istruzione riservata al sistema operativo.
Uso improprio dei dati	Parte dei dati è del tipo sbagliato, o non è inizializzata.
Intervento dell'operatore o del sistema operativo	Per qualche regione l'operatore o il sistema operativo ha fatto terminare il processo (per esempio, in caso di stallo).
Terminazione del genitore	Quando un genitore termina, il sistema operativo può automaticamente far terminare tutti i discendenti di quel genitore.
Richiesta del genitore	Un processo genitore ha l'autorità di far terminare uno dei suoi discendenti

l'allocatore non può selezionare il processo da più tempo in coda, ma deve percorrere la lista, cercando il processo che non sia bloccato e che sia da più tempo in coda.

Una soluzione più efficace consiste nel suddividere lo stato di non-esecuzione in due stati: Ready e Blocked (Figura 3.5). Per completezza, sono stati aggiunti altri due stati, di cui vedremo in seguito l'utilità. Questi sono i cinque stati nel nuovo diagramma:

- **Running**: un processo che è correntemente in esecuzione. In questo capitolo prendiamo in esame un computer con un singolo processore, pertanto può trovarsi in questo stato non più di un processo alla volta.
- **Ready**: un processo pronto per l'esecuzione.
- **Blocked**: un processo che non può essere eseguito sino al verificarsi di un certo evento, ad esempio il completamento di un'operazione di I/O.
- **New**: un processo appena creato, ma ancora non ammesso dal sistema operativo nell'insieme dei processi eseguibili.
- **Exit**: un processo che è stato rilasciato dall'insieme dei processi eseguibili, o perché è terminato, o perché è fallito per una qualche ragione.

Gli stati New ed Exit sono costrutti utili per la gestione dei processi. New corrisponde ad un processo che è stato appena definito: se un nuovo utente cerca ad esempio di entrare in un sistema time-sharing, o un nuovo job batch è presentato per l'esecuzione, il sistema operativo può costruire un nuovo processo in due passi. Dapprima, effettua i necessari compiti routinari di manutenzione: associa un identificatore al processo, alloca e costruisce le tabelle necessarie per la gestione del processo stesso. A questo punto il processo è nello stato New, in quanto il sistema operativo ha effettuato le azioni necessarie per crearlo, ma non si è impegnato ad eseguirlo. Infatti, il sistema operativo può limitare il numero dei processi presenti nel sistema, per non limitarne le prestazioni, o per non sovraccaricare la memoria.

Allo stesso modo, un processo esce dal sistema in due passi. Un processo termina quando raggiunge il suo completamento, quando fallisce a causa di errori non recuperabili, o quando un altro processo, provvisto degli opportuni privilegi, provoca la sua terminazione. La terminazione determina la transizione allo stato di Exit, e a questo punto il processo non può più essere scelto per l'esecuzione. Tuttavia, il sistema operativo conserva temporaneamente le tabelle e le

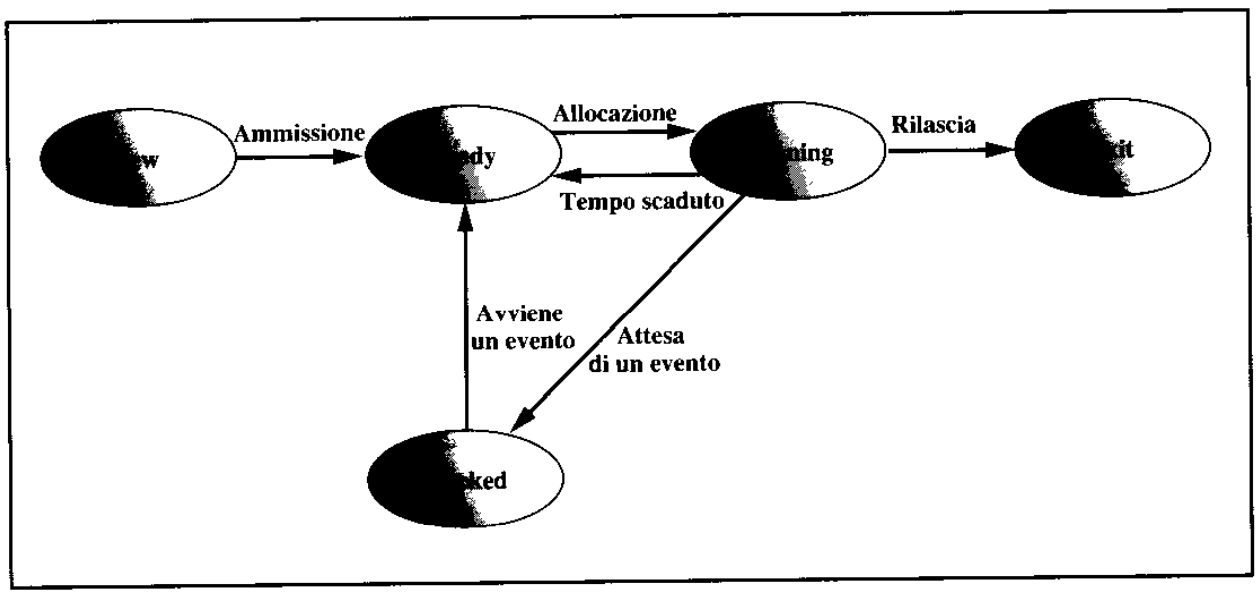


Figura 3.5 *Modello di processi a cinque stati*

altre informazioni associate al job, in modo che programmi ausiliari o di supporto possano estrarne le informazioni eventualmente necessarie: per esempio, un programma per la contabilità delle risorse può registrare il tempo di processore e le altre risorse al fine di fatturare un servizio, mentre un programma di utilità può aver bisogno di informazioni sulla storia del processo, per analizzare le prestazioni o l'utilizzo delle risorse. Una volta che questi programmi hanno estratto le informazioni di cui hanno bisogno, il sistema operativo non deve più mantenere i dati relativi al processo, che è cancellato.

La Figura 3.5 presenta la tipologia degli eventi che conducono a ciascuna transizione di stato. Le possibili transizioni sono:

- **Null → New**: un nuovo processo è creato per eseguire un programma; l'evento avviene per uno dei motivi elencati nella Tabella 3.1.
- **New → Ready:** il sistema operativo sposta un processo da New a Ready quando prende in carico un nuovo processo. La maggior parte dei sistemi pone un limite, in base al numero di processi già esistenti, oppure alla quantità di memoria virtuale ad essi destinata; lo scopo di questo limite è garantire che il numero dei processi attivi non sia tale da degradare le prestazioni.
- **Ready → Running**: quando è tempo di selezionare un nuovo processo per l'esecuzione, il sistema operativo ne sceglie uno nello stato Ready. Il problema della scelta di questo processo è affrontato nella Parte Quarta.
- **Running → Exit**: il processo correntemente in esecuzione è fatto terminare dal sistema operativo se il processo stesso indica di aver terminato, o se fallisce.
- **Running → Ready**: questa transizione è di solito motivata dal fatto che il processo ha raggiunto il tempo massimo permesso per un'esecuzione ininterrotta; in pratica, tutti i sistemi operativi multiprogrammati impongono questa forma di organizzazione del tempo. Esistono poi diverse altre cause per questa transizione, che non sono però implementate in tutti i sistemi operativi. Ad esempio, se il sistema operativo assegna diversi livelli di priorità a diversi processi, un processo può essere prerilasciato: supponiamo che il processo A sia in esecuzione a un dato livello di priorità, e che il processo B, ad un livello di priorità più alto, sia bloccato. Quando il sistema operativo si accorge che l'evento atteso dal processo B è avvenuto, passa B allo stato di Ready, e può interrompere A, mandando in esecuzione B. Ricordiamo inoltre che un processo può volontariamente rilasciare il controllo del processore.
- **Running → Blocked**: un processo va nello stato Blocked, se sta aspettando una risorsa. La richiesta di una risorsa al sistema operativo viene di solito effettuata in forma di una chiamata di servizio al sistema (cioè una chiamata dal programma in esecuzione ad una procedura che è parte del codice sistema operativo). Ad esempio, un processo può richiedere al sistema operativo un servizio che quest'ultimo non è preparato ad eseguire immediatamente, oppure una risorsa (un file, una sezione condivisa della memoria virtuale) non immediatamente disponibile; è possibile che il processo inizi un'azione, ad esempio un'operazione di I/O, che va completata prima che il processo stesso possa continuare, o che sia bloccato in attesa dell'input o di un messaggio da parte di un altro processo, nel caso comunichi con questo.

Tabella 3.3 *Stati dei processi per la traccia di esecuzione della Figura 3.3.*

Periodi di tempo	Processo A	Processo B	Processo C
1-6	**Running**	Ready	Ready
7-12	Ready	Ready	Ready
13-16	Ready	**Running**	Ready
17-22	Ready	Blocked	Ready
23-28	Ready	Blocked	**Running**
29-34	Ready	Blocked	Ready
35-40	**Running**	Blocked	Ready
41-46	Ready	Blocked	Ready
47-52	Ready	Blocked	**Running**

- **Blocked → Ready**: un processo nello stato Blocked passa a quello di Ready quando accade l'evento che stava aspettando.
- **Ready → Exit**: per chiarezza questa transizione non compare nel diagramma di stato. In molti sistemi un processo genitore può far terminare un figlio in ogni momento; inoltre, se un genitore termina, tutti i processi figli associati al genitore potrebbero terminare.
- **Blocked → Exit**: valgono le osservazioni del caso precedente.

Ritornando al nostro semplice esempio, la Tabella 3.3. illustra le transizioni di ciascun processo fra i cinque stati. La Figura 3.6a suggerisce il modo in cui potrebbe essere implementata una strategia di accodamento: ci sono ora due code, una dei Ready e una dei Blocked. Quando un processo è ammesso nel sistema, è posto nella coda dei Ready, e sempre da questa coda il sistema sceglie un nuovo processo per l'esecuzione. In assenza di uno schema di priorità, si può utilizzare semplicemente una coda first-in-first-out: un processo in esecuzione, secondo le circostanze, può successivamente essere posto nella coda dei Ready o dei Blocked, o terminare.

Infine, quando accade un evento, perché tutti i processi nella coda dei Blocked che ne sono in attesa passino alla coda dei Ready, il sistema operativo deve scandire l'intera coda dei Blocked, cercando i processi in attesa di quell'evento. In un grande sistema operativo, in quella coda potrebbero trovarsi centinaia o migliaia di processi: sarebbe assai più efficiente disporre di diverse code, una per ciascun tipo di evento, in modo da portare allo stato di Ready l'intera lista dei processi, presente nella coda appropriata, al verificarsi dell'evento atteso (Figura 3.6b).

È possibile migliorare ulteriormente: se la scelta dei processi è stabilita secondo uno schema a priorità, è conveniente avere diverse code dei Ready, una per ciascun livello di priorità. Il sistema potrebbe così determinare velocemente quale sia il processo pronto a più elevata priorità e in attesa da più lungo tempo.

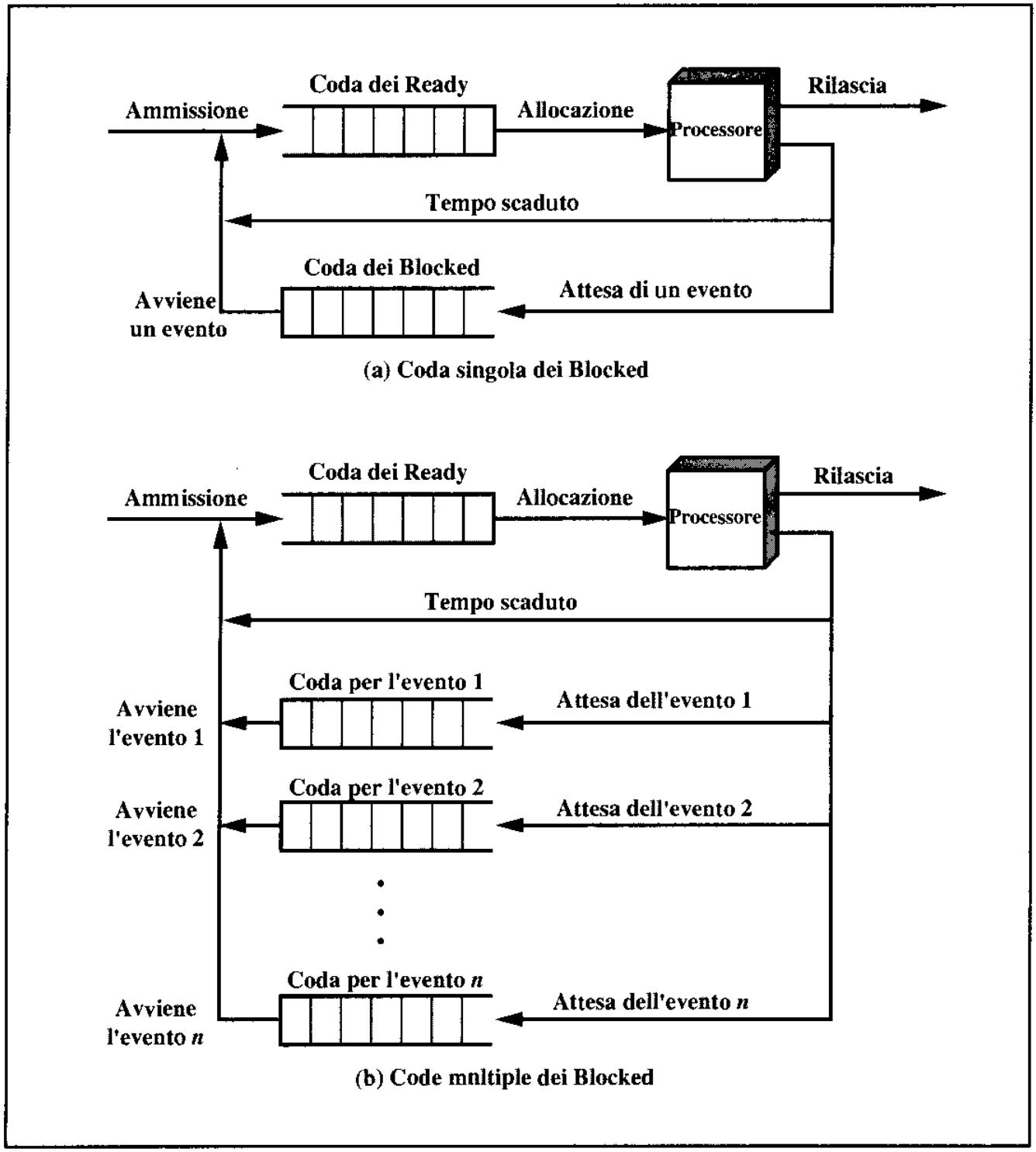


Figura 3.6 *Modello di accodamento riferito alla Figura 3.5*

Processi sospesi

La necessità dello swapping

I tre stati principali appena descritti (Ready, Running, Blocked), permettono di costruire un modello sistematico del comportamento dei processi e di guidare l'implementazione dei sistemi operativi, molti dei quali sono costruiti utilizzando proprio questi tre stati.

Esiste tuttavia una valida giustificazione per aggiungere altri stati al modello: per apprezzarne i vantaggi, si consideri un sistema privo di memoria virtuale. Ciascun processo, per essere eseguito, deve essere totalmente caricato nella memoria principale: nella Figura 3.6b tutti i processi in tutte le code dovrebbero risiedere nella memoria principale.

Ricordiamo però che tutto questo elaborato congegno è causato dalla lentezza delle attività di I/O rispetto all'elaborazione, e pertanto il processore, in un sistema monoprogrammato, resta inattivo per la maggior parte del tempo. La soluzione proposta dalla Figura 3.6b non risolve però del tutto il problema. È vero che in questo caso la memoria contiene molti processi, ed il processore può passare ad un altro processo, se quello corrente è in attesa; il processore, però, è tanto più veloce delle operazioni di I/O, che tutti i processi in memoria resteranno di norma in attesa di operazioni di I/O: anche con la multiprogrammazione, il processore rischia di restare inattivo la maggior parte del tempo.

Quali potrebbero essere le soluzioni? Ad esempio, espandere la memoria principale per accogliere più processi; tale approccio presenta però due punti deboli. Primo: la memoria principale ha un costo che, per quanto piccolo per ogni bit, diventa ragguardevole per Megabit o Gigabit. Secondo: la fame di memoria dei programmi è cresciuta tanto rapidamente quanto è sceso il costo della memoria stessa, col risultato che attualmente memorie più grandi ospitano processi più grandi, ma non più numerosi.

Un'altra soluzione è lo swapping, che consiste nello spostare un processo, o una sua parte, dalla memoria principale al disco: quando nessun processo in memoria principale si trova nello stato Ready, il sistema operativo trasferisce uno dei processi bloccati sul disco nella coda dei *Suspend*, dove si trovano quei processi temporaneamente espulsi dalla memoria principale, in altre parole sospesi. Successivamente, il sistema operativo carica un altro processo dalla coda dei Suspend, o risponde ad una richiesta di un processo nello stato New. L'esecuzione continua poi con il processo appena arrivato.

Questa è in ogni modo un'operazione di I/O, e quindi esiste il rischio di peggiorare la situazione, piuttosto che di migliorarla, ma poiché l'I/O del disco è generalmente quello più veloce nel sistema (in confronto ad un nastro o ad una stampante), lo swapping di norma migliorerà le prestazioni.

Con l'operazione di trasferimento dei processi su disco appena descritta, dobbiamo aggiungere al nostro modello del comportamento dei processi (Figura 3.7a) lo stato Suspend: quando tutti i processi nella memoria principale sono nello stato Blocked, il sistema operativo può sospendere un processo, ponendolo nello stato Suspend e trasferendolo sul disco. Lo spazio rimasto libero nella memoria principale può essere utilizzato per caricare un altro processo.

Quando il sistema operativo ha effettuato l'operazione di *scaricamento sul disco* (*swap out*), per scegliere il processo da caricare in memoria principale ha due possibilità: ammettere un processo appena creato, oppure uno precedentemente sospeso. A prima vista, sembrerebbe più vantaggioso caricare un processo sospeso per servire le sue richieste, piuttosto che incrementare il carico totale sul sistema, ma quest'opzione presenta una difficoltà. Tutti i processi sospesi, infatti, erano nello stato Blocked al momento della sospensione: chiaramente, non trarremmo alcun beneficio dal ricaricare di nuovo in memoria principale un processo bloccato, perché non è ancora pronto per l'esecuzione. Tuttavia, ciascun processo sospeso era originariamente bloccato su un particolare evento: quando tale evento si verifica, il processo non è più bloccato ed è potenzialmente disponibile per l'esecuzione.

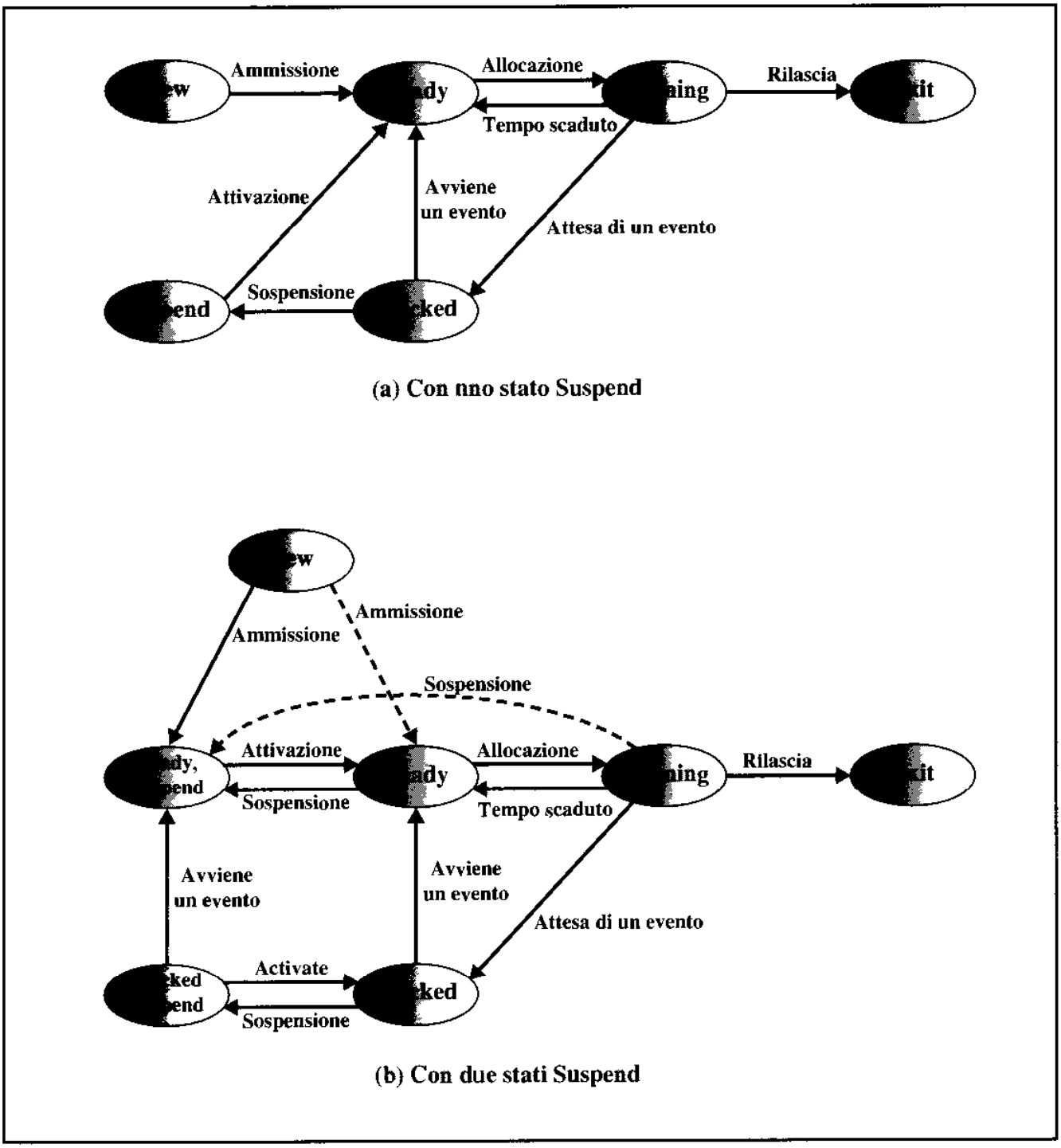


Figura 3.7 *Diagramma di transizione degli stati, con gli stati Suspend*

Occorre quindi ripensare questo aspetto della progettazione: siamo di fronte a due casi indipendenti: un processo è in attesa di un evento (bloccato o no), un processo è stato scaricato dalla memoria principale (sospeso o no). Per sistemare questa combinazione 2 x 2 sono necessari quattro stati:

- **Ready:** il processo in memoria principale è disponibile per l'esecuzione.
- **Blocked:** il processo è in memoria principale e in attesa di un evento.

- **Blocked-Suspend:** il processo è in memoria secondaria e in attesa di un evento.
- **Ready-Suspend**: il processo è in memoria secondaria ma è disponibile per l'esecuzione appena caricato in memoria principale.

Prima di esaminare il diagramma di transizione di stato che comprende i due nuovi stati Suspend, dovremmo toccare un altro argomento. Fino ad ora, abbiamo presupposto che la memoria virtuale non fosse in uso, e che un processo si trovasse per intero in memoria principale, o interamente fuori di essa. Con uno schema di memoria virtuale, però, è possibile eseguire un processo solo parzialmente contenuto nella memoria principale: se c'è un riferimento ad un indirizzo che non è in memoria principale, è possibile caricare l'appropriata porzione del processo. L'uso della memoria virtuale sembrerebbe eliminare la necessità di un trasferimento esplicito, poiché qualsiasi indirizzo, in qualsiasi processo, può essere trasferito dentro o fuori dalla memoria principale dall'hardware di gestione della memoria del processore. Tuttavia, come vedremo nel Capitolo 8, le prestazioni di un sistema a memoria virtuale possono crollare, se esiste un numero sufficientemente grande di processi attivi, tutti parzialmente presenti in memoria principale; quindi, anche in un sistema di memoria virtuale, per mantenere un buon livello delle prestazioni, il sistema operativo dovrà di tanto in tanto effettuare espressamente un trasferimento completo dei processi.

Guardiamo ora al nostro modello di transizione di stato (Figura 3.7b; le linee tratteggiate indicano transizioni possibili ma non necessarie), che comprende ora nuove e importanti transizioni:

- **Blocked → Blocked-Suspend**: in assenza di processi pronti, è trasferito su disco almeno un processo bloccato, per fare spazio ad un altro che non lo è. Questa transizione è possibile anche se esistono processi pronti disponibili, qualora il sistema operativo stabilisca che il processo corrente, o un processo pronto che vorrebbe selezionare, richiedano più memoria principale per mantenere prestazioni adeguate.
- **Blocked-Suspend → Ready-Suspend**: tale transizione avviene al verificarsi dell'evento per cui il processo era stato sospeso. Ciò richiede che le informazioni di stato sui processi sospesi siano accessibili al sistema operativo.
- **Ready-Suspend → Ready**: quando non ci sono in memoria principale processi pronti, il sistema operativo dovrà caricarne uno per continuare l'esecuzione. È possibile, inoltre, che un processo nello stato Ready-Suspend abbia priorità più alta dei processi nello stato Ready. In questo caso il progettista può stabilire che sia più importante considerare il processo a maggior priorità, piuttosto che minimizzare i trasferimenti su disco.
- **Ready → Ready-Suspend:** di norma, il sistema operativo preferirebbe sospendere un processo bloccato piuttosto che uno pronto, poiché quello pronto può essere eseguito subito, mentre uno bloccato sta occupando spazio di memoria principale e non può essere eseguito. Tuttavia, può essere necessario sospendere un processo pronto, se solo così si riesce a liberare un blocco sufficientemente grande di memoria principale. Inoltre, il sistema operativo può sospendere un processo pronto a bassa priorità, piuttosto che un processo bloccato a priorità maggiore, se ritiene che il processo bloccato sarà pronto fra breve.

Esistono altre transizioni utili da tenere in considerazione:

- **New → Ready-Suspend e New → Ready**: un nuovo processo appena creato può essere aggiunto alla coda dei Ready oppure a quella dei Ready-Suspend. In entrambi i casi, il sistema operativo deve costruire delle tabelle per gestire il processo ed allocare il relativo spazio degli indirizzi: il sistema operativo potrebbe preferire di effettuare presto queste operazioni, in modo da mantenere un ampio insieme di processi non bloccati. Con questa strategia, però, lo spazio nella memoria principale sarebbe spesso insufficiente per un nuovo processo; di qui, l'uso della transizione New → Ready-Suspend. D'altro canto, con una filosofia just-in-time di creazione dei processi il più tardi possibile, si riduce il sovraccarico del sistema, che riesce a creare nuovi processi, pur essendo affollato di processi bloccati.

- **Blocked-Suspend → Blocked**: questa transizione può sembrare inutile: se un processo non è pronto per l'esecuzione e non si trova già in memoria principale, perché caricarvelo? Consideriamo però un'eventualità di questo genere: un processo termina liberando memoria principale; un processo, nella coda Blocked-Suspend, ha priorità maggiore di tutti i processi della coda Ready-Suspend, e il sistema operativo ha ragione di credere che l'evento bloccante di quel processo accadrà presto. In queste circostanze, appare ragionevole caricare un processo bloccato in memoria principale, preferendolo ad uno pronto.

- **Running → Ready-Suspend:** di norma, un processo in esecuzione passa allo stato Ready quando termina il quanto di tempo che gli era stato destinato. Se però il sistema operativo sta scegliendo un processo (a più alta priorità e appena sbloccato) dalla coda dei Blocked-Suspend, potrebbe spostare il processo in esecuzione direttamente alla coda dei Ready-Suspend, liberando memoria principale.

- **Tutti → Exit**: di regola, un processo in esecuzione termina, se è giunto alla fine o se ha incontrato qualche condizione fatale d'errore. Tuttavia, in qualche sistema operativo, un processo può essere fatto terminare dal processo che lo ha creato, o quando termina il processo genitore stesso. Se ciò è possibile, un processo può passare allo stato di Exit da qualunque altro stato.

Altri usi della sospensione

Fin qui abbiamo considerato equivalenti i concetti di processo sospeso e di processo non contenuto in memoria principale. Un processo che non è in memoria principale non è immediatamente disponibile per l'esecuzione, sia che aspetti un evento, sia in caso contrario.

È possibile generalizzare il concetto di processo sospeso. Definiamo ora un processo dotato delle seguenti caratteristiche:

1. Non è immediatamente disponibile per l'esecuzione.

2. Può essere in attesa di un evento, o no. Se è in attesa, la condizione di Blocked è indipendente da quella di Suspend, e il verificarsi dell'evento bloccante non abilita il processo all'esecuzione.

3. Era stato posto nello stato Suspend da un agente (se stesso, un processo genitore oppure il sistema operativo) al fine di impedire la sua esecuzione.

4. Non può essere rimosso da questo stato finché l'agente non ordina esplicitamente la rimozione.

La Tabella 3.4 elenca alcune cause della sospensione di un processo. Una fra queste, che abbiamo già esaminato, è la necessità di trasferire un processo sul disco per consentire di caricare un processo pronto, o semplicemente per diminuire la pressione sul sistema di memoria virtuale, in modo che gli altri processi abbiano a disposizione più memoria principale. Il sistema operativo può sospendere un processo anche per altre ragioni. Ad esempio, nel caso di un processo per il controllo o per la registrazione di **tracce**, impiegato per controllare l'attività del sistema; tale processo registra il livello di utilizzo di diverse risorse (processore, memoria, canali) e il tasso di avanzamento dei processi utente nel sistema; il sistema operativo, sotto il controllo dell'operatore, può attivare o disattivare questo processo di volta in volta. Il sistema sospende un processo, se individua un problema o ne sospetta l'esistenza (come nel caso dello stallo, esaminato nel Capitolo 6). Un altro motivo potrebbe essere un problema su una linea di comunicazione: in questo caso, l'operatore deve far sì che il sistema operativo sospenda il processo che sta utilizzando la linea durante l'esecuzione dei test.

Altri motivi per sospendere un processo riguardano le azioni degli utenti interattivi. Ad esempio, se un utente sospetta un errore nel programma, deve effettuarne il debug, sospendendone l'esecuzione, esaminando e modificando il programma o i dati, per poi riprendere l'esecuzione. Oppure, l'utente può voler attivare o disattivare un processo in background, che sta raccogliendo statistiche di contabilità o tracciati di esecuzione.

Si può decidere di trasferire un processo su disco anche per problemi relativi alla temporizzazione: se un processo deve essere attivato periodicamente, ma per la maggior parte del tempo resta inattivo, si dovrebbe scaricarlo su disco nei periodi di inattività (pensiamo, come esempio, ad un programma per il controllo dell'uso delle risorse o dell'attività utente).

Tabella 3.4 *Cause della sospensione dei processi*

Swap (trasferimento su disco)	Il sistema operativo deve liberare sufficiente memoria principale per caricare un processo pronto per l'esecuzione.
Altre cause ascrivibili al sistema operativo	Il sistema operativo può sospendere un processo in background, o di utilità, o un processo sospettato di causare un problema.
Richiesta di un utente interattivo	Un utente può voler sospendere l'esecuzione di un programma per effettuare il debug o quando viene usata una risorsa.
Temporizzazione	Un processo può essere eseguito periodicamente (ad es., un processo per il monitoraggio del sistema o per la contabilità) e può essere sospeso mentre aspetta l'intervallo di tempo successivo.
Richiesta del processo genitore	Un processo genitore può voler sospendere l'esecuzione di un discendente per esaminare o modificare i processi sospesi o per coordinare l'attività dei diversi discendenti.

Infine, un processo genitore può voler sospendere un processo figlio. Per esempio, il processo A genera B per leggere un file; quindi B, nella procedura di lettura del file, incontra un errore e lo riferisce al processo A; A sospende B per trovare le cause dell'errore.

In tutti questi casi, l'attivazione di un processo sospeso è richiesta dall'agente che inizialmente aveva determinato la sospensione.

3.2 Descrizione dei processi

Il sistema operativo controlla gli eventi che avvengono nel sistema di elaborazione: schedula e seleziona i processi per l'esecuzione, alloca le relative risorse e risponde alle richieste dei programmi utente per i servizi di base. Si può con ragione affermare che il sistema operativo sia l'entità che gestisce l'uso delle risorse di sistema da parte dei processi.

Questo concetto è illustrato nella Figura 3.8; in un ambiente multiprogrammato ci sono diversi processi (P_1, ..., P_n), che sono stati creati ed esistono nella memoria virtuale; ciascun processo, durante la sua esecuzione, ha bisogno di accedere a certe risorse di sistema, tra le quali il processore, i dispositivi di I/O e la memoria principale. Nella figura, P_1 è in esecuzione; almeno una parte del processo è in memoria principale e controlla due dispositivi di I/O. Anche il processo P_2 è in memoria principale, ma è bloccato in attesa del dispositivo di I/O assegnato a P_1. P_n è stato scaricato su disco e quindi sospeso.

La gestione di queste risorse da parte del sistema operativo per conto dei processi sarà oggetto di un esame dettagliato nei prossimi capitoli; per ora, cercheremo di rispondere ad una domanda fondamentale: di quali informazioni ha bisogno il sistema operativo per controllare i processi e gestire per conto loro le risorse?

Strutture di controllo del sistema operativo

Per gestire processi e risorse, il sistema operativo ha bisogno di informazioni riguardo al loro stato corrente e, per averle disponibili, procede di norma in modo semplice: costruisce e mantie-

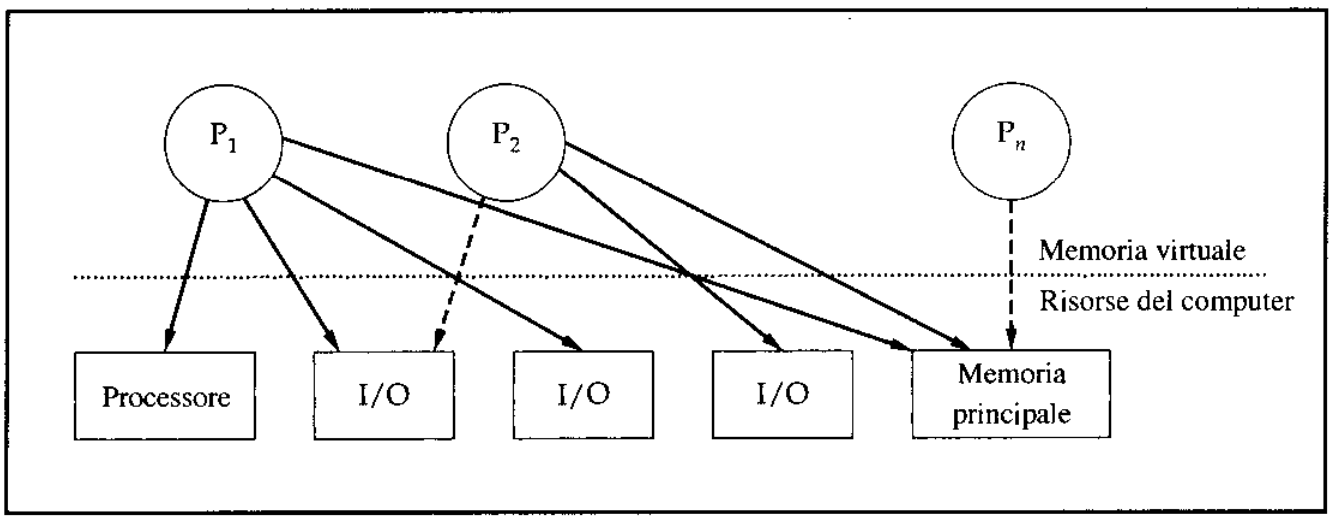


Figura 3.8 *Processi e risorse*

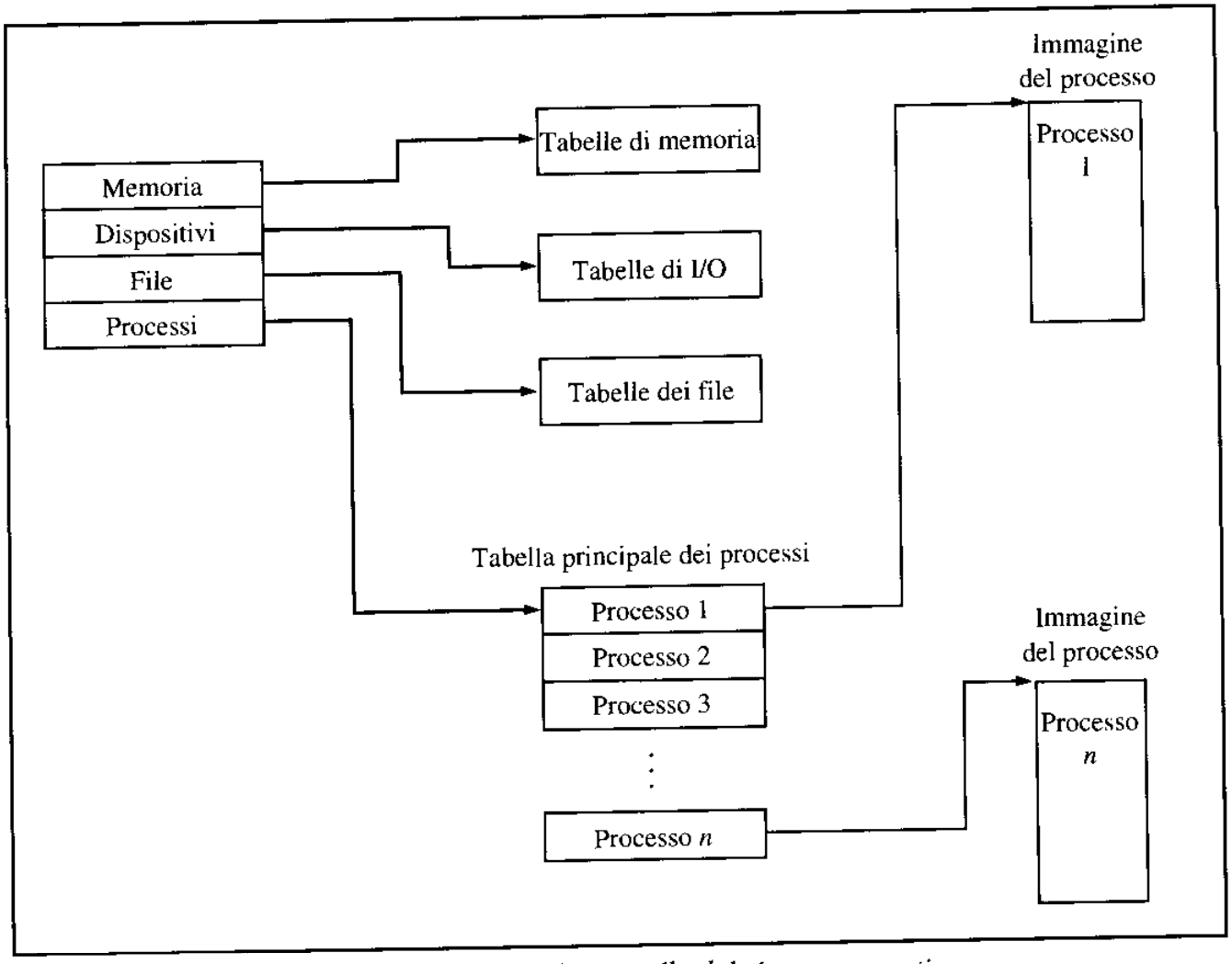


Figura 3.9 *Struttura generale delle tabelle di controllo del sistema operativo*

ne tabelle di informazioni relative a ciascuna delle entità che gestisce. La Figura 3.9 suggerisce un'idea generale dell'estensione di questo compito, presentando quattro diversi tipi di tabelle gestite dal sistema operativo: memoria, I/O, file, processi. Benché i vari sistemi operativi non coincidano nei dettagli, fondamentalmente tutti raccolgono le informazioni in queste quattro categorie.

Le **tabelle di memoria** conservano traccia sia della memoria principale (reale) che di quella secondaria (virtuale). Parte della memoria principale è riservata al sistema operativo, e il resto agli altri processi. I processi sono mantenuti in memoria secondaria utilizzando una memoria virtuale o un semplice meccanismo di swapping. Le tabelle di memoria contengono le seguenti informazioni:

- Allocazione della memoria principale ai processi.
- Allocazione della memoria secondaria ai processi.
- Attributi di protezione dei segmenti della memoria principale o di quella virtuale (ad esempio, quali processi possano accedere a determinate regioni di memoria condivise).
- Informazioni necessarie per gestire la memoria virtuale.

Studieremo in dettaglio le strutture per la gestione della memoria nella Parte Terza.

Il sistema operativo utilizza le **tabelle di I/O** per gestire i dispositivi di I/O e i canali del sistema di elaborazione. A un dato momento, un dispositivo di I/O può essere disponibile, oppure assegnato ad un particolare processo; se un'operazione di I/O è in esecuzione, il sistema operativo deve conoscerne lo stato e la locazione della memoria principale utilizzata come sorgente o destinazione del trasferimento di I/O (la gestione dell'I/O sarà esaminata nel Capitolo 11).

Il sistema operativo gestisce anche le **tabelle dei file**, che forniscono informazioni sull'esistenza dei file, la loro locazione in memoria secondaria, il loro stato corrente ed altri loro attributi. Gran parte di queste informazioni, se non tutte, possono essere mantenute ed utilizzate da un sistema per la gestione dei file, nel qual caso il sistema operativo ha una conoscenza ridotta o nulla dell'esistenza dei file. In altri casi, la gestione dei file è demandata interamente al sistema operativo stesso; di quest'argomento ci occuperemo nel Capitolo 12.

Il sistema operativo, infine, deve anche mantenere delle tabelle per la gestione dei processi, e a queste **tabelle dei processi** sono dedicate le pagine restanti di questa sezione. Prima di inoltrarci nella trattazione, sono però necessarie due precisazioni: benché la Figura 3.9 presenti i quattro insiemi di tabelle distinti, è ovvio che le tabelle dovrebbero essere collegate in qualche modo, o consentire riferimenti incrociati. La memoria, l'I/O e i file sono gestiti per conto dei processi, e quindi le tabelle dei processi devono contenere, direttamente o indirettamente, dei riferimenti a queste risorse. I file referenziati dalle tabelle dei file sono accessibili attraverso un dispositivo di I/O e si troveranno talvolta in memoria principale o virtuale; le tabelle stesse devono essere accessibili al sistema operativo, e quindi sono soggette alla gestione della memoria.

Il secondo punto si riferisce a quanto abbiamo detto prima, in altre parole che è il sistema operativo a creare e mantenere queste tabelle. La prima domanda che dovremmo porci riguarda dunque il modo in cui il sistema operativo crea inizialmente le tabelle. Com'è ovvio, gli occorrono alcune informazioni fondamentali sul sistema, ad esempio quanta memoria esista, quali siano i dispositivi di I/O, quali sono i loro identificatori, e così via. Si tratta di un problema di configurazione: quando il sistema operativo è inizializzato, deve poter accedere ad alcune informazioni di configurazione, e questi dati devono essere creati al di fuori del sistema operativi, con l'intervento e l'assistenza di un operatore umano.

Strutture di controllo dei processi

Precisiamo dunque di quali informazioni debba disporre il sistema operativo per gestire e controllare un processo: in primo luogo, deve sapere dove è memorizzato il processo e, in secondo luogo, conoscere gli attributi del processo necessari per la sua gestione.

Locazione dei processi

Prima di trattare della locazione di un processo o di quali siano i suoi attributi, è necessario però affrontare un problema ancor più fondamentale: come appare fisicamente un processo? In una forma minima, un processo contiene un programma, o un insieme di programmi, che devono essere eseguiti. Associato a tali programmi, esiste un insieme di locazioni, per i dati, per le

variabili globali e locali, e per ogni costante definita; quindi, un processo comprende almeno una memoria sufficiente per mantenere i suoi programmi ed i suoi dati. Inoltre, l'esecuzione di un programma comprende di regola uno stack (vedi l'Appendice 1B), usato per conservare traccia delle chiamate di procedura e dei parametri passati fra le procedure; per finire, a ciascun processo sono associati diversi attributi, utilizzati dal sistema operativo per il controllo dei processi stessi. Tipicamente, la collezione degli attributi è definita come **process control block** (blocco di controllo del processo) [4]. Quest'insieme di programma, dati, stack e attributi è definita con il termine **immagine del processo** (Tabella 3.5).

La locazione di un'immagine del processo dipenderà dallo schema di gestione della memoria in uso: nel caso più semplice, l'immagine del processo è mantenuta come blocco contiguo in memoria secondaria, usualmente sul disco. Affinché il sistema possa gestire il processo, almeno una piccola porzione della sua immagine deve essere mantenuta nella memoria principale; per eseguire il processo, la totalità della sua immagine deve essere caricata in memoria principale, o almeno in quella virtuale. Di conseguenza, il sistema operativo deve conoscere la posizione di ciascun processo sul disco e nella memoria principale (per quelli in essa presenti). Una variante più complessa di questo schema è stata analizzata parlando del sistema CTSS (Capitolo 2): in tale sistema, quando un processo è scaricato sul disco, parte della sua immagine può rimanere nella memoria principale, quindi il sistema operativo deve conservare traccia di quali porzioni dell'immagine di ciascun processo si trovino ancora in memoria principale.

La maggior parte dei sistemi operativi moderni utilizza uno schema di gestione della memoria in cui un'immagine del processo si compone di un insieme di blocchi, non necessariamente contigui. A seconda dello schema utilizzato, questi blocchi se hanno lunghezza variabile sono chiamati segmenti, se invece sono di lunghezza fissa sono chiamati pagine; è anche possibile una combinazione delle due forme. In ogni caso, tali schemi permettono al sistema operativo di caricare solo una porzione di un particolare processo: ad un momento dato, una porzione di

Tabella 3.5 *Elementi tipici di un'immagine del processo*

Dati utente La parte modificabile dello spazio utente. Può includere i dati del programma, un'area stack utente e programmi che possono essere modificati.
Programma utente Il programma che dev'essere eseguito.
Stack di sistema Ciascun processo ha uno o più stack di sistema LIFO associati. Uno stack è utilizzato per memorizzare parametri e indirizzi per le chiamate di procedura di sistema.
Process control block Dati che servono al sistema operativo per controllare i processi (vedi Tabella 3.6)

[4] Altri nomi comunemente usati per questa struttura sono task control block (blocco di controllo del task), descrittore del processo e descrittore del task.

immagine del processo può essere in memoria principale, e la parte restante in quella secondaria[5]. Quindi, le tabelle dei processi mantenute dal sistema operativo devono mostrare la locazione di ciascun segmento e/o pagina di ciascun'immagine del processo.

La Figura 3.9 illustra la struttura delle informazioni relative alla posizione delle immagini dei processi: c'è una tabella principale dei processi con una entry per ciascun processo, ognuna delle quali contiene almeno un puntatore all'immagine; se tale immagine contiene blocchi multipli, questa informazione si trova direttamente nella tabella principale dei processi, oppure è disponibile tramite riferimenti incrociati alle tabelle di memoria. Naturalmente, questa è una descrizione generale: ogni sistema operativo, infatti, organizza queste informazioni in modi diversi.

Attributi dei processi

Un sistema sofisticato di multiprogrammazione richiede una grande quantità di informazioni riguardo a ciascun processo. Come già sappiamo, questa informazione risiede in un process control block, e sistemi diversi la organizzano in modi diversi, come vedremo negli esempi alla fine di questo capitolo e anche nel prossimo. Per ora, analizziamo semplicemente il tipo di informazione che potrebbe essere usata da un sistema operativo, senza considerare in dettaglio come sia organizzata.

La Tabella 3.6 elenca le tipiche categorie di informazioni richieste dal sistema operativo per ciascun processo. La quantità di tali informazioni potrebbe sorprenderci, ma quando conosceremo più a fondo i compiti di un sistema operativo, questo lungo elenco ci apparirà motivato.

Si possono raggruppare le informazioni del process control block in tre categorie generali:

- Identificazione del processo
- Informazioni sullo stato del processore
- Informazioni di controllo del processo.

Quanto all'**identificazione del processo**, praticamente in tutti i sistemi operativi a ciascun processo è assegnato un unico identificatore numerico, che può essere semplicemente un indice nella tabella primaria dei processi (Tabella 3.9); altrimenti, deve esistere una mappatura che permetta al sistema operativo di localizzare le tabelle appropriate basandosi sull'identificatore del processo, che è utile in diversi modi; molte delle altre tabelle controllate dal sistema operativo usano, infatti, gli identificatori dei processi per riferirsi alle tabelle dei processi stessi. Ad esempio, le tabelle di memoria possono essere organizzate per fornire una mappa della memoria principale che indichi quale processo sia assegnato a ciascuna regione; riferimenti simili appaiono nelle tabelle di I/O e dei file. Quando i processi comunicano fra di loro, l'identificatore del

[5] Questa breve analisi tralascia alcuni dettagli. In particolare, la totalità dell'immagine relativa ad un processo attivo è sempre in memoria secondaria; quando una parte dell'immagine è caricata in memoria principale, è copiata piuttosto che trasferita. Quindi, la memoria secondaria contiene una copia di tutti i segmenti e/o pagine: se la porzione in memoria principale è modificata, la copia in memoria secondaria resterà non aggiornata finché la parte in memoria principale non sarà copiata sul disco.

Tabella 3.6 *Elementi tipici di un Process Control Block*

Identificazione del processo

Identificatori

Gli identificatori numerici che possono essere memorizzati in un process control block comprendono

- Identificatore di questo processo
- Identificatore del processo che ha creato questo processo (processo genitore)
- Identificatore utente.

Informazioni sullo stato del processore

Registri visibili all'utente

I registri visibili all'utente sono quelli cui si può accedere attraverso il linguaggio macchina eseguito dal processore. Di solito sono presenti da 8 a 32 di questi registri, benchè alcune implementazioni RISC ne abbiano più di 100.

Registri di controllo e di stato

Esistono diversi registri del processore utilizzati per controllare le operazioni del processore, e comprendono:

- *Program counter*: contiene l'indirizzo dell'istruzione successiva da prelevare.
- *Codici di condizione*: impostati dalle più recenti operazioni aritmetiche o logiche (segno, zero, riporto, uguale, overflow).
- *Informazioni di stato*: includono i flag abilitato o disabilitato delle interruzioni e la modalità di esecuzione.

Puntatori allo stack

Ciascun processo ha uno o più stack di sistema LIFO associati. Uno stack è utilizzato per memorizzare parametri e indirizzi per le chiamate di procedura di sistema. Il puntatore dello stack punta alla cima dello stack.

Informazioni di controllo del processo

Schedulazione e informazioni di stato

Queste sono le informazioni che servono al sistema operativo per effettuare la schedulazione.

Gli elementi tipici delle informazioni sono:

- *Stato del processo*: definisce la disponibilità del processo ad essere schedulato per l'esecuzione (Running, Ready, Blocked, Halt).
- *Priorità*: uno o più campi per descrivere la priorità di schedulazione del processo possono essere usati. In alcuni sistemi sono richiesti diversi valori (per difetto, corrente, la più alta disponibile).
- *Informazione correlata alla schedulazione*: dipende dall'algoritmo di schedulazione usato, ad esempio, il tempo di attesa del processo e la durata del tempo di esecuzione l'ultima volta che è stato eseguito.
- *Evento*: identità dell'evento che il processo sta aspettando prima di poter essere ripreso.

Strutturazione dei dati

Un processo può essere collegato ad altri in una coda, in un anello, o in qualche altra struttura. Ad esempio, tutti i processi in uno stato di attesa di un particolare livello di priorità possono essere collegati in una coda. Un processo può avere una relazione genitore-figlio (creatore-creato) con un altro processo. Il process control block può contenere puntatori ad altri processi per supportare queste strutture.

Tabella 3.6 *Continua*

Comunicazione fra i processi
Diversi flag, segnali e messaggi possono essere associati con la comunicazione fra due processi indipendenti. Parte o tutta questa informazione può essere mantenuta nel process control block.

Privilegi dei processi
Sono concessi privilegi ai processi in relazione alla memoria cui possono accedere ed ai tipi di istruzioni che possono eseguire. Inoltre, i privilegi si possono applicare all'uso delle funzioni di utilità del sistema ed ai servizi.

Gestione della memoria
Questa sezione può contenere puntatori a segmenti e/o a pagine che descrivono la memoria virtuale assegnata al processo.

Proprietà e utilizzo delle risorse
Memorizza le risorse controllate dal processo, come i file aperti. La storia dell'utilizzo del processore o di altre risorse può inoltre essere presenta; questa informazione può essere usata dallo scheduler.

processo informa il sistema operativo della destinazione di una particolare comunicazione; quando si permette ai processi di crearne altri, gli identificatori indicano il genitore e i discendenti di ciascun processo.

In aggiunta a questi identificatori, può essere assegnato ad un processo un identificatore dell'utente responsabile del job.

Le **informazioni sullo stato del processore** comprendono i contenuti dei registri del processore. Mentre un processo è in esecuzione, naturalmente, le informazioni sono contenute nei registri, ma quando è interrotto, tutte le informazioni dei registri devono essere salvate, in modo da poterle ripristinare alla ripresa dell'esecuzione del processo. La natura e il numero dei registri coinvolti dipendono dall'architettura del processore: tipicamente, l'insieme dei registri comprende i registri visibili all'utente, i registri di controllo e di stato e i puntatori allo stack (di cui si è parlato nel Capitolo 1).

In particolare, tutte le architetture di processore prevedono un registro o un insieme di registri, spesso noti come program status word (PSW, o parola di stato del processore), che contengono codici e altre informazioni di stato. Un buon esempio di PSW è quello presente sulle macchine Pentium, detto registro EFLAGS (Figura 3.10 e Tabella 3.7), utilizzato dai sistemi

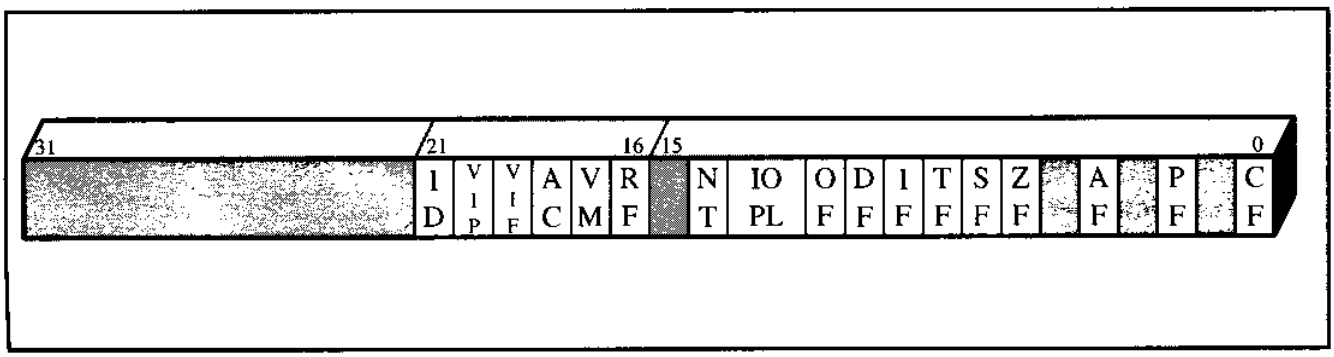


Figura 3.10 *Registro EFLAGS del Pentium*

Tabella 3.7 *Bit del Registro EFLAGS del Pentium*

Bit di controllo

AC (Controllo dell'allineamento)
Impostato se una parola semplice o doppia è disallineata rispetto ai limiti di parola semplice o doppia

ID (Flag di Identificazione)
Se questo bit può essere impostato o azzerato, significa che questo processore supporta l'istruzione CPUID. Questa istruzione fornisce informazioni sul produttore, la famiglia ed il modello

RF (Flag di ripristino)
Permette al programmatore di disabilitare le eccezioni di debug in modo che l'istruzione possa ripartire dopo una eccezione di debug senza causare immediatamente un'altra eccezione di debug.

IOPL (Livello di privilegio di I/O)
Quando impostato, causa la generazione di un'eccezione su tutti gli accessi ai dispositivi di I/O durante le operazioni in modalità protetta.

DF (Flag di direzione)
Determina se le istruzioni per l'elaborazione delle stringhe incrementano o decrementano i semi-registri a 16-bit SI ed DI (per le operazioni a 16 bit) o i registri a 32 bit ESI ed EDI (per le operazioni a 32 bit).

IF (Flag di abilitazione delle interruzioni)
Quando impostato, il processore riconosce la interruzioni esterne

TF (Flag di trap)
Quando impostato, causa un'interruzione dopo l'esecuzione di ciascuna istruzione. Usato per il debug

Bit dei modi operativi

NT (Flag dei task annidati)
Indica che il task corrente è annidato entro un altro task in modalità operativa protetta

VM (Modo virtuale 8086)
Permette al programmatore di abilitare e disabilitare il modo 8086 virtuale, che determina se il processore esegue come una macchina 8086.

Codici di condizione

AF (Flag di riporto ausiliario)
Rappresenta il riporto o il prestito fra i semi-byte di un'operazione aritmetica o logica a 8 bit che utilizza il registro AL

CF (Flag di riporto)
Indica il riporto o il prestito nella posizione del bit più a sinistra in segnito ad un'operazione aritmetica

OF (Flag di overflow)
Indica un overflow aritmetico dopo un'addizione o una sottrazione

PF (Flag di parità)
Parità del risultato di un'operazione aritmetica o logica

SF (Flag di segno)
Indica il segno del risultato di un'operazione logica o aritmetica

ZF (Flag di zero)
Indica che il risultato di un'operazione logica o aritmetica è 0.

operativi (inclusi UNIX e Windows NT) eseguiti sui computer Pentium.

La terza categoria di informazioni del process control block possiamo definirla, in mancanza di un nome migliore, **informazioni di controllo del processo**: si tratta di informazioni supplementari, necessarie al sistema operativo per controllare e ordinare i vari processi attivi. L'ultima parte della Tabella 3.6 presenta le caratteristiche di queste informazioni: quando (nei prossimi capitoli) studieremo nei dettagli il modo di funzionare del sistema operativo, capiremo perché siano necessari i vari elementi che compongono questo lista.

La Figura 3.11 suggerisce la struttura delle immagini dei processi nella memoria virtuale; ciascun'immagine si compone di un process control block, dello stack dell'utente, dello spazio privato degli indirizzi del processo, e di ogni altro spazio di indirizzamento che il processo condivide con altri. Nella figura, ciascun'immagine di processo appare come un intervallo contiguo di indirizzi: in un'implementazione reale, potrebbe non essere così, in relazione allo schema di gestione della memoria e al modo in cui le strutture di controllo sono organizzate dal sistema operativo.

Come indicato nella Tabella 3.6, il process control block può contenere informazioni per la strutturazione dei dati, ad esempio i puntatori che permettono il collegamento dei process control block: quindi, le code descritte nella sezione precedente potrebbero essere implementate come liste di process control block. Ad esempio, le strutture di code della Figura 3.6a potrebbero essere implementate come suggerito dalla Figura 3.12.

Il ruolo del process control block

Il process control block è la struttura dati più importante del sistema operativo, in quanto ciascun process control block contiene tutte le informazioni necessarie sui processi. I blocchi sono letti e/o modificati in pratica da ogni modulo del sistema operativo, compresi quelli per la schedulazione, l'allocazione delle risorse, l'elaborazione delle interruzioni, il controllo e l'analisi delle prestazioni.

Si può ben dire che l'insieme dei process control block definisce lo stato del sistema operativo e costituisce un problema importante nella progettazione. Alle informazioni contenute nei process control block devono accedere diverse routine del sistema operativo, e fornire loro un accesso diretto a queste tabelle non è difficile: ciascun processo, infatti, è dotato di un unico ID (identificatore), che può essere utilizzato come un indice nella tabella dei puntatori ai process control block. La difficoltà non consiste dunque nell'accesso, ma piuttosto nella protezione. Ci sono, infatti, due problemi:

- Un baco in una sola routine (ad esempio, un gestore delle interruzioni) potrebbe danneggiare i process control block, distruggendo la capacità del sistema di gestire i processi coinvolti.
- Un cambiamento nella struttura o nella semantica del process control block potrebbe coinvolgere diversi moduli del sistema operativo.

Questi problemi si possono affrontare richiedendo che tutte le routine del sistema operativo passino attraverso una routine di gestione, il cui solo compito consista nel proteggere i process control block, e che sia l'unica con il permesso di leggere e scrivere questi blocchi. L'uso di una

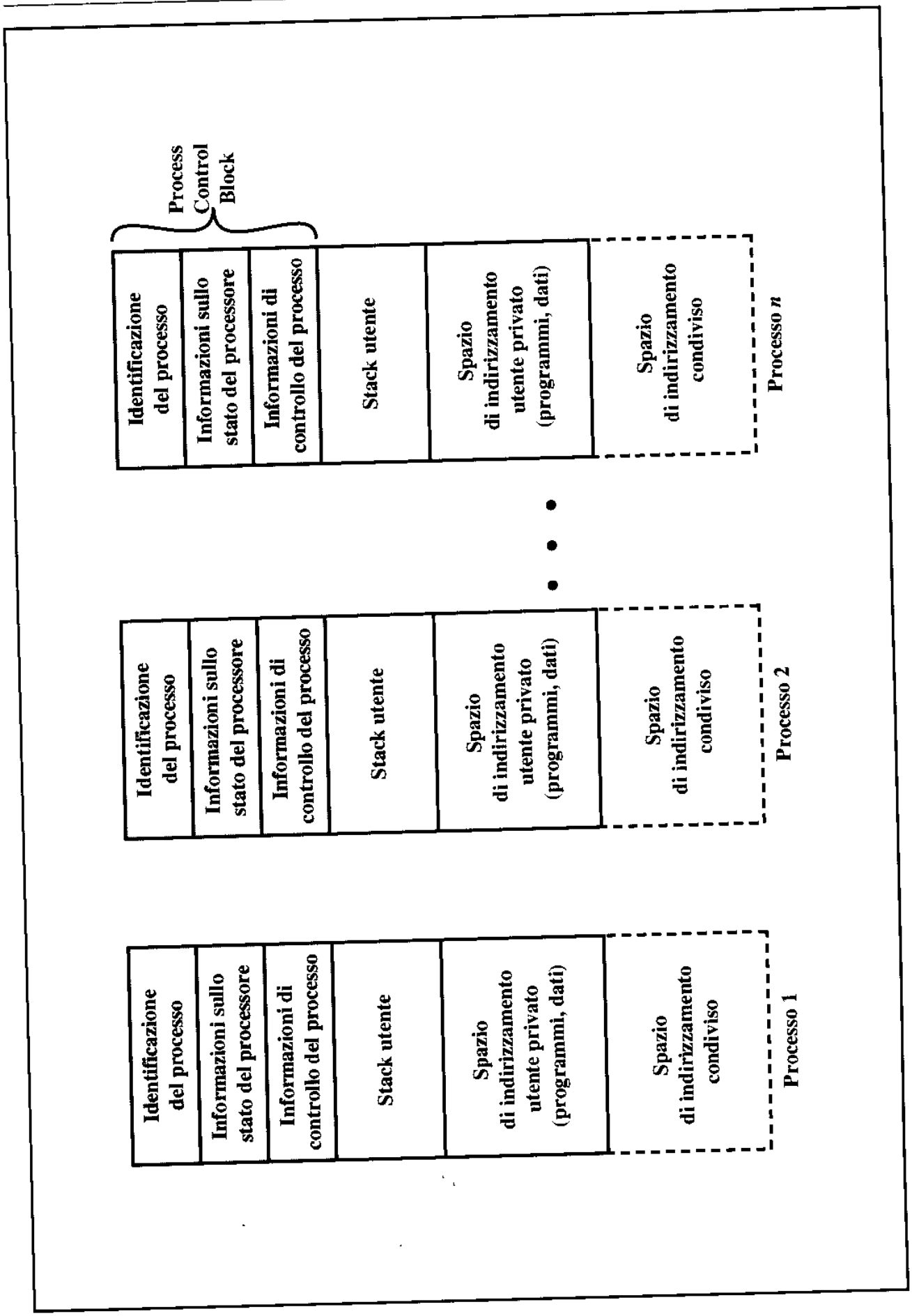


Figura 3.11 *Processi utente in memoria virtuale*

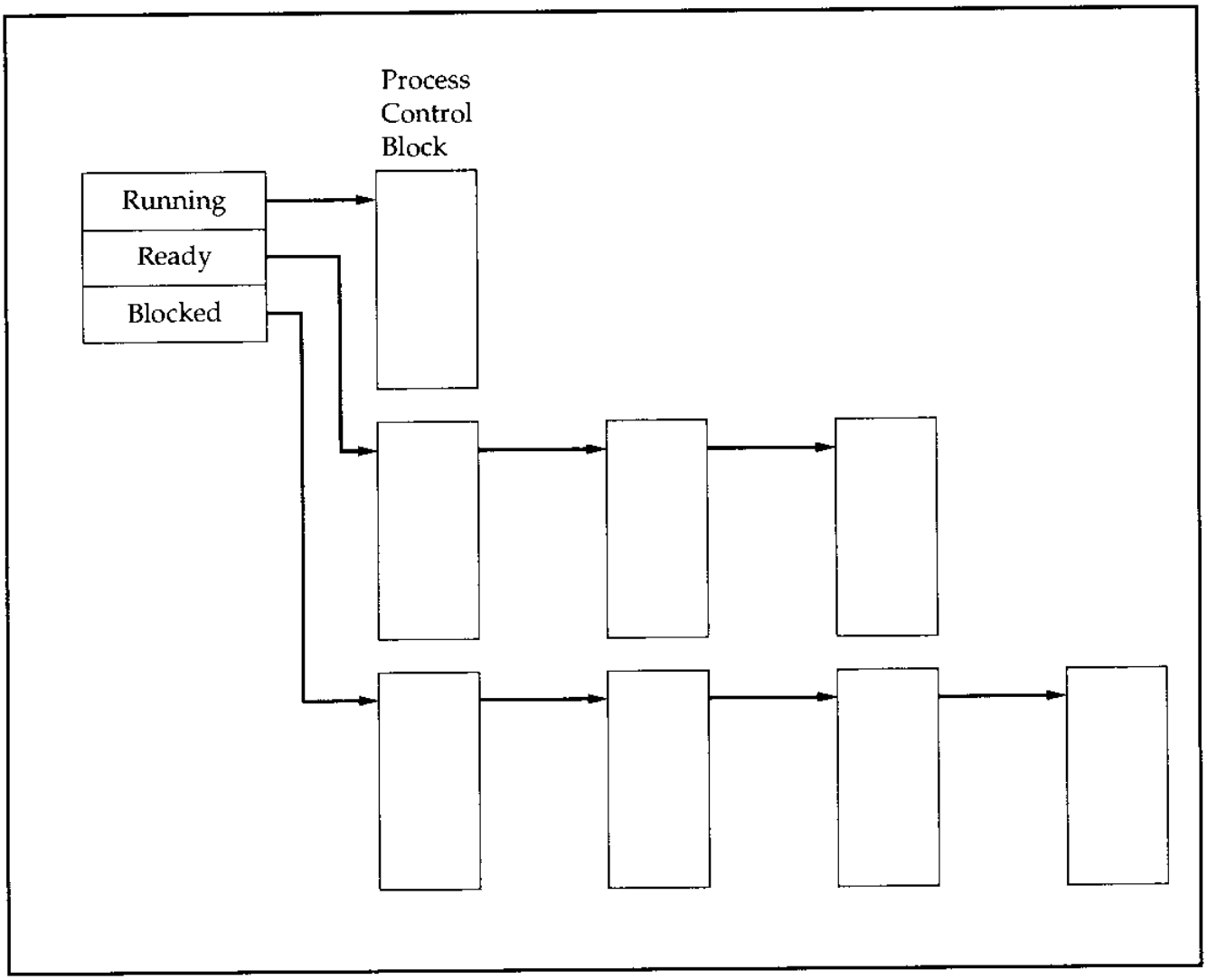


Figura 3.12 *Liste dei processi*

tale routine tuttavia richiede un compromesso fra le prestazioni globali del sistema e il grado di affidabilità e correttezza del resto del software di sistema.

3.3 Controllo dei processi

Modi di esecuzione

Prima di continuare l'esame della gestione dei processi da parte del sistema operativo, occorre distinguere tra il modo di esecuzione del processore, normalmente associato al sistema operativo, e quello normalmente associato ai programmi utente. La maggior parte dei processori supporta almeno due modi di esecuzione, e certe istruzioni si possono eseguire solo nel modo più privilegiato, ad esempio la lettura o la scrittura di un registro di controllo, o della program status word, le istruzioni primitive di I/O e quelle relative alla gestione della memoria. A certe regioni della memoria, inoltre, è possibile accedere solo in modalità più privilegiata.

Il modo meno privilegiato spesso è definito modo utente, in quanto di norma i programmi utenti sono eseguiti in tale modalità. Il modo privilegiato, invece, è chiamato modo di sistema, di controllo, oppure modo kernel; quest'ultima definizione si riferisce al nucleo del sistema operativo, ovvero a quella sua parte che comprende le funzioni di sistema più importanti. La Tabella 3.8 elenca le funzioni tipicamente contenute nel kernel di un sistema operativo.

Dovrebbe essere chiaro perché si usino due modi di esecuzione: è necessario, infatti, proteggere il sistema operativo e le tabelle principali del sistema, come il process control block, dalle interferenze dei programmi utente. In modo kernel, il software ha un completo controllo del processore, di tutte le sue istruzioni, dei registri e della memoria; tale livello di controllo per i programmi utente non è necessario e, per motivi di sicurezza, è preferibile che essi non ne usufruiscano.

Sorgono ora due problemi: come fa il processore a sapere quale sia il suo modo di esecuzione corrente? Come può cambiarlo? Riguardo al primo quesito, di solito esiste un bit nella PSW che indica il modo di esecuzione, e che è cambiato in risposta a determinati eventi; ad esempio, quando un utente effettua una chiamata ad un servizio del sistema operativo, la modalità passa al modo kernel: operazione tipicamente effettuata eseguendo un'istruzione che cambia il modo. Un esempio è fornito dall'istruzione Change Mode (CHM) del VAX: quando un utente effettua una chiamata di sistema, oppure quando un'interruzione trasferisce il controllo ad una routine di sistema, questa esegue una CHM per entrare in modalità più privilegiata, e la esegue di nuovo per entrare in una modalità meno privilegiata, prima di restituire il controllo al processo utente.

Tabella 3.8 *Funzioni tipiche di un kernel di sistema operativo*

GESTIONE DEI PROCESSI

- Creazione e terminazione dei processi
- Schedulazione e allocazione dei processi
- Cambio di processi
- Sincronizzazione dei processi e supporto per la comunicazione fra i processi
- Gestione dei process control block

GESTIONE DELLA MEMORIA

- Allocazione di uno spazio di indirizzi ai processi
- Trasferimento da memoria a disco e viceversa (swap)
- Gestione della paginazione e della segmentazione

GESTIONE DELL'I/O

- Gestione dei buffer
- Allocazione dei canali di I/O e dei dispositivi per i processi

FUNZIONI DI SUPPORTO

- Gestione delle interruzioni
- Contabilità
- Controllo

Se un programma utente cerca di eseguire una CHM, si avrà semplicemente una chiamata al sistema operativo, che ritornerà un errore, a meno che il cambio di modo non sia permesso.

Creazione dei processi

Nella Sezione 3.1 abbiamo esaminato gli eventi che conducono alla creazione di un nuovo processo. Avendo passato in rassegna anche le strutture dati associate ai processi, ora possiamo descrivere brevemente i passi necessari alla loro creazione.

Una volta che il sistema operativo, per una qualsiasi ragione, decide di creare un nuovo processo, può proseguire in questo modo:

1. Assegnare al nuovo processo un identificatore unico, ed aggiungere una nuova entry alla tabella principale dei processi, che contiene una entry per ogni processo.
2. Allocare spazio per il processo, comprendendo tutti gli elementi dell'immagine del processo: il sistema deve sapere quanto spazio sia necessario per gli indirizzi privati del processo utente (programmi e dati) e per lo stack utente. Questi valori possono essere assegnati per difetto, in base al tipo di processo, oppure determinati su richiesta dell'utente, al momento della creazione del job. Se un processo è creato da un altro, il genitore può passare i valori necessari al sistema operativo come parte della richiesta di creazione del processo figlio; se uno spazio di indirizzi già esistente deve essere condiviso con questo nuovo processo, occorre approntare gli opportuni collegamenti. Infine, deve essere allocato lo spazio per il process control block.
3. Il process control block dev'essere inizializzato: la parte di identificazione del processo contiene l'ID di questo processo, più altri ID appropriati, ad esempio quello del processo genitore. La parte di informazioni sullo stato del processore viene di norma inizializzata con la maggior parte dei valori a zero, eccetto il program counter (impostato al punto di entrata del programma) e i puntatori allo stack di sistema (impostati in modo da definire i limiti dello stack del processo). La parte relativa alle informazioni di controllo del processo è inizializzata basandosi su valori standard per difetto e sugli attributi richiesti per questo processo; ad esempio, lo stato sarà inizializzato a Ready, oppure Ready-Suspend. La priorità può essere impostata per difetto al valore più basso, a meno che non esista una richiesta esplicita di una priorità maggiore. Inizialmente, il processo può non possedere alcuna risorsa (dispositivi di I/O, file), a meno che non le abbia ereditate dal genitore, oppure abbia fatto una richiesta esplicita.
4. Occorre impostare collegamenti appropriati; ad esempio, se il sistema operativo mantiene le code di schedulazione come liste, il nuovo processo dev'essere posto nella coda dei Ready, o dei Ready-Suspend.
5. È possibile dover creare o estendere altre strutture dati; ad esempio, il sistema può mantenere un file di contabilità per ciascun processo, utile in seguito per la fatturazione e/o la valutazione delle prestazioni.

Cambio dei processi

In apparenza, la funzione di cambio (*switch*) di processo (il cambiamento del processo correntemente in esecuzione con un altro scelto dallo schedulatore) sembrerebbe banale: ad un certo punto, un processo in esecuzione è interrotto e il sistema operativo assegna ad un altro lo stato Running, passandogli il controllo. Sorgono a questo proposito diversi problemi di progettazione: riconoscere quali eventi suscitino un cambio di processo, distinguere tra quest'ultimo e il cambio di modalità di esecuzione, determinare quali operazioni sulle proprie strutture dati debba effettuare il sistema per realizzare un cambio di processo.

Quando effettuare il cambio di processo

Un cambio di processo può verificarsi ogni volta che il sistema operativo ottiene il controllo dal processo correntemente in esecuzione. La Tabella 3.9 suggerisce gli eventi che possono riportare il controllo al sistema operativo.

Esaminiamo in primo luogo le interruzioni di sistema; in effetti, è possibile distinguerne (come fanno molti sistemi) due tipi, uno dei quali è semplicemente detto interruzione mentre l'altro è chiamato trap. Il primo è dovuto a determinati eventi, esterni e indipendenti rispetto al processo correntemente in esecuzione, ad esempio il completamento di un'operazione di I/O. Il secondo è relativo ad un errore, o ad una condizione d'eccezione generata entro il processo corrente (un tentativo illegale di accesso ad un file). Con un'**interruzione** ordinaria, il controllo prima passa ad un gestore dell'interruzione, che effettua alcune operazioni di manutenzione di base e poi salta ad una routine del sistema operativo, relativa al particolare tipo di interruzione verificatosi. Ecco alcuni esempi:

- **Interruzione di clock**: il sistema operativo determina se il processo correntemente in esecuzione sia stato eseguito per il quanto di tempo massimo permesso. In caso affermativo, il processo passa allo stato Ready e ne è selezionato un altro.
- **Interruzione di I/O**: il sistema operativo determina quale operazione di I/O sia avvenuta. Nel caso si tratti di un evento che uno o più processi attendono, il sistema operativo li passa

Tabella 3.9 *Meccanismi per interrompere l'esecuzione di un processo*

Meccanismo	Causa	Uso
Interruzione	Esterna rispetto all'esecuzione dell'istruzione corrente	Reazione ad un evento esterno asincrono
Trap	Associata all'esecuzione dell'istruzione corrente	Gestione di un errore o di una condizione eccezionale
Chiamata supervisore	Richiesta esplicita	Chiamata ad una funzione del sistema operativo

dallo stato Blocked a Ready (e i processi Blocked-Suspend passano allo stato Ready-Suspend), quindi decide se riprendere l'esecuzione del processo correntemente nello stato Running, oppure passare a un altro processo Ready, con priorità più elevata.

- **Fault di memoria**: il processore incontra un riferimento a un indirizzo di memoria virtuale, per una parola che non si trova in memoria principale. Il sistema operativo carica il blocco (pagina o segmento) di memoria contenente il riferimento, dalla memoria secondaria a quella principale; soddisfatta la richiesta di caricamento, che è un'operazione di I/O, il sistema operativo effettua un cambio per riprendere l'esecuzione di un altro processo: il processo che ha generato il fault di memoria è posto nello stato Blocked e, caricato in memoria il blocco richiesto, quel processo passa allo stato di Ready.

Con una **trap** il sistema operativo determina se un errore sia fatale o meno. In caso affermativo il processo correntemente in esecuzione passa allo stato di Exit e avviene un cambio di processo, altrimenti, l'azione del sistema operativo dipende dalla natura dell'errore o dalle scelte di progettazione: può tentare una procedura di recupero, o limitarsi a notificare il fatto all'utente, può effettuare un cambio di processo, oppure riprendere il processo correntemente in esecuzione.

Infine, anche una **chiamata a supervisore** da parte del programma in esecuzione può attivare il sistema operativo: un processo utente in esecuzione può effettuare una richiesta di I/O (l'apertura di un file), e questa chiamata trasferisce il controllo ad una routine, che è parte del sistema operativo. Di norma, l'uso di una chiamata di sistema comporta il passaggio del processo utente allo stato Blocked.

Cambio di modo di esecuzione

Nel Capitolo 1 abbiamo esaminato il ciclo di interruzione come parte del ciclo di istruzione. Ricordiamo che, nel ciclo di interruzione, il processore controlla se sia avvenuta un'interruzione, indicata da un apposito segnale; se non ci sono interruzioni pendenti, il processore continua con il ciclo di fetch e preleva l'istruzione successiva del programma corrente; in caso contrario, effettua le seguenti operazioni:

1. Salva il contesto del programma correntemente in esecuzione.
2. Imposta il program counter all'indirizzo di partenza di un programma di *gestione delle interruzioni.*
3. Effettua il passaggio da modalità utente a modalità kernel, in modo che l'elaborazione dell'interruzione possa contenere istruzioni privilegiate.

Il processore continua con il ciclo di fetch e preleva la prima istruzione del programma di gestione delle interruzioni, che servirà l'interruzione stessa.

A questo punto, dobbiamo chiederci da che cosa sia costituito il contesto che occorre salvare: esso deve contenere le informazioni che possono essere alterate dall'esecuzione del gestore dell'interruzione, e che saranno necessarie per ripristinare il programma interrotto. Occorre sal-

vare, dunque, quella parte del process control block definita come informazioni sullo stato del processore, che comprende il program counter, gli altri registri del processore e le informazioni relativa allo stack.

Occorrono ancora altre operazioni? La risposta dipende da ciò che avviene in seguito. Il gestore dell'interruzione solitamente è un piccolo programma che svolge pochi compiti di base: modifica il flag o l'indicatore che segnala la presenza di un'interruzione, può mandare un acknowledgment (*segnale di risposta*) all'entità che ha generato l'interruzione (un modulo di I/O), e può effettuare alcune operazioni di manutenzione di base, relative agli effetti dell'evento che ha causato l'interruzione. Ad esempio, se l'interruzione si riferisce ad un evento di I/O, il gestore controllerà se si è verificata una condizione di errore e, in caso affermativo, invierà un segnale al processo che ha richiesto l'operazione di I/O. Se invece l'interruzione proviene dal clock, il gestore passa il controllo all'allocatore, il quale a sua volta lo ripassa ad un altro processo, poiché il quanto di tempo assegnato al processo correntemente in esecuzione è terminato.

Riguardo alle altre informazioni contenute nel process control block, se l'interruzione è seguita da un passaggio ad un altro processo, saranno necessarie altre operazioni. Tuttavia, la parola chiave dell'affermazione precedente è il *se:* nella maggior parte dei sistemi operativi, infatti, il verificarsi di un'interruzione non comporta necessariamente un cambio di processo: dopo che il gestore dell'interruzione è andato in esecuzione, il processo correntemente in esecuzione può riprendere. In questo caso, è sufficiente salvare le informazioni sullo stato del processore al momento dell'interruzione, e ripristinarla quando il controllo è restituito al programma che era in esecuzione. Tipicamente, le funzioni di salvataggio e ripristino sono effettuate in hardware.

Cambiamento dello stato di un processo

Dovrebbe essere ormai chiaro che cambio di modo e cambio di processo sono due concetti ben distinti[6]. Un cambio di modo può avvenire senza cambiare lo stato del processo correntemente in esecuzione: il salvataggio e ripristino del contesto comportano in questo caso solo un piccolo sovraccarico. Al contrario, se il processo correntemente in esecuzione passa ad un altro stato (Ready, Blocked ecc.), il sistema operativo deve effettuare cambiamenti sostanziali nel proprio ambiente. Questi sono i passi necessari ad effettuare un completo cambio di processo:

1. Salvataggio del contesto del processore, compresi il program counter e gli altri registri.
2. Aggiornamento del process control block del processo correntemente nello stato Running, compreso il cambiamento dello stato del processo (Ready; Blocked; Ready-Suspend; Exit). Occorre aggiornare anche altre informazioni rilevanti, tra le quali il motivo per cui è stato abbandonato lo stato Running, oppure le informazioni di contabilità.
3. Spostamento del process control block nella coda appropriata (Ready; Blocked sull'evento *i*; Ready-Suspend).

[6] Il termine *cambio di contesto* compare spesso nei manuali e nella letteratura sui sistemi operativi. Sfortunatamente, benché di solito questo termine designi ciò che qui definiamo cambio di processo, in altri testi indica il cambio di modalità, o anche un cambio di thread (che vedremo nel capitolo seguente). Per evitare ambiguità, in questo testo il termine non compare.

4. Selezione di un altro processo per l'esecuzione (di cui tratterà la Parte Quarta).
5. Aggiornamento del process control block del processo selezionato, che comprende il passaggio allo stato Running.
6. Aggiornamento delle strutture dati per la gestione della memoria, che dipende da come è gestita la traduzione degli indirizzi (vedi Parte Terza).
7. Ripristino del contesto del processore relativo al processo che aveva precedentemente abbandonato lo stato Running, ricaricando i vecchi valori del program counter e degli altri registri.

Quindi, il cambio di processo, quando comporta un cambio di stato, richiede operazioni più numerose di un semplice cambio di modo.

Esecuzione del sistema operativo

Nel Capitolo 2, a proposito dei sistemi operativi, abbiamo sottolineato due elementi degni di attenzione:

- Il sistema operativo funziona come il software ordinario, essendo un programma eseguito dal processore.
- Il sistema operativo cede frequentemente il controllo e, per poterlo riacquistare, dipende dal processore.

Se il sistema operativo altro non è che una collezione di programmi, ed è eseguito dal processore proprio come ogni altro programma, potremmo definirlo un processo? In caso affermativo, come è controllato? A queste interessanti domande, i progettisti di sistemi operativi hanno fornito varie risposte. La Figura 3.13 illustra alcuni approcci, caratteristici di diversi sistemi operativi contemporanei.

Kernel non implementati con processi

Un approccio, piuttosto tradizionale e comune in molti sistemi operativi meno recenti, consiste nell'esecuzione del kernel al di fuori di qualsiasi processo (Figura 3.13a): quando il processo correntemente in esecuzione è interrotto o effettua una chiamata al supervisore, il suo contesto di modo è salvato ed il controllo passa al kernel. Il sistema operativo ha la sua propria regione di memoria ed il suo proprio stack di sistema per controllare le chiamate ed i ritorni da procedura; può eseguire ogni funzione desiderata e ripristinare il contesto del processo interrotto, determinando la ripresa dell'esecuzione nel processo utente interrotto. In alternativa, il sistema operativo può completare l'operazione di salvataggio dell'ambiente del processo e procedere poi a schedulare e sceglierne un altro. Ciò può accadere a seconda della causa dell'interruzione e delle circostanze del momento.

In ogni caso, risulta evidente che il concetto di processo si considera applicabile solo ai programmi utente, in quanto il codice del sistema operativo è eseguito come un'entità separata, che opera in modalità privilegiata.

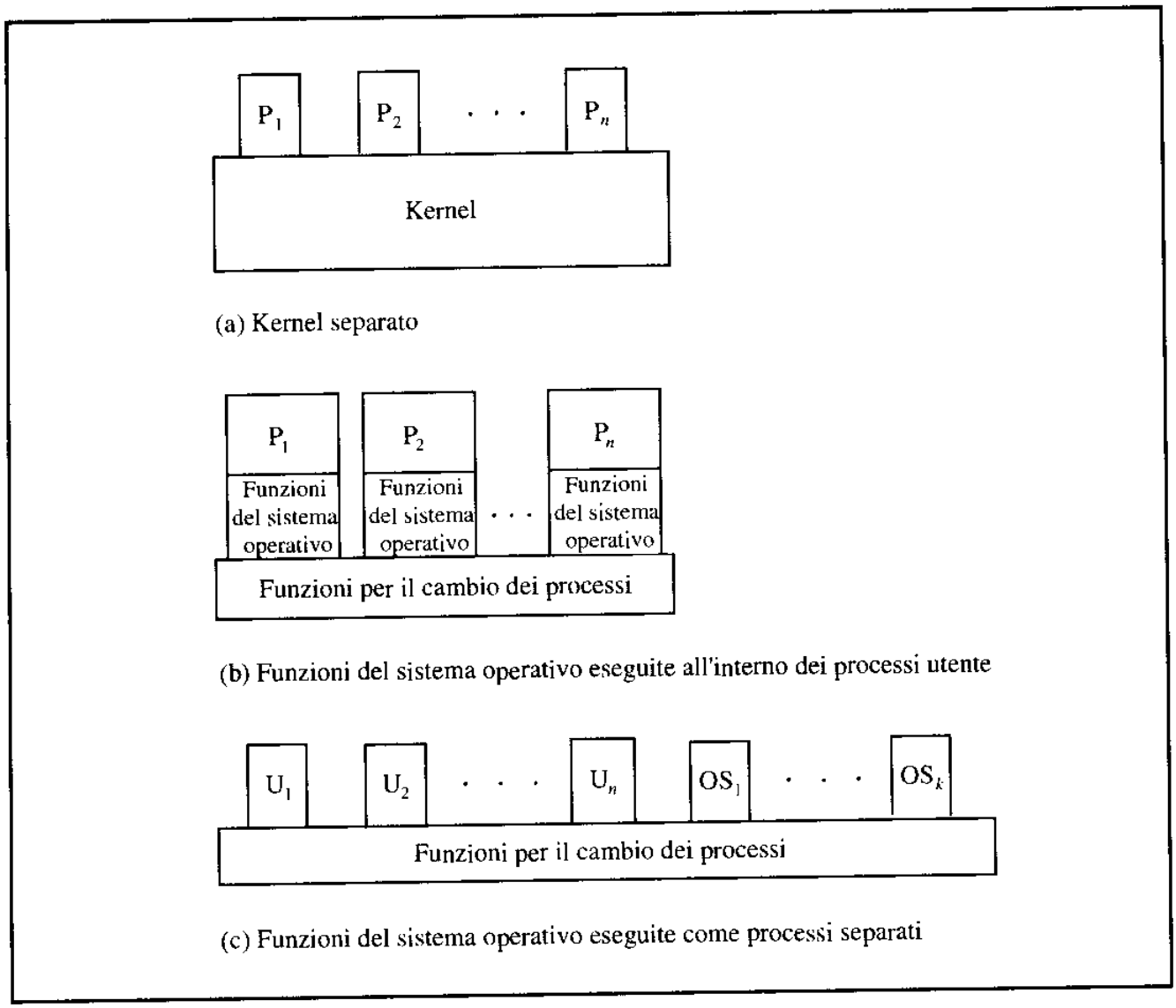


Figura 3.13 *Relazioni tra sistema operativo e processi utente*

Esecuzione all'interno dei processi utente

Un'altra possibilità, comune in sistemi operativi di macchine più piccole (mini e microcomputer) consiste nell'eseguire quasi tutto il software del sistema operativo nel contesto di un processo utente. In questo caso, il sistema operativo è visto in primo luogo come un insieme di routine che l'utente chiama per effettuare diverse funzioni, eseguite all'interno dell'ambiente del processo utente (Figura 3.13b). Ad un dato momento, il sistema operativo sta gestendo *n* immagini dei processi, e ciascuna contiene non solo le regioni mostrate nella Figura 3.11, ma anche programmi, dati e aree dello stack per i programmi kernel.

La Figura 3.14 suggerisce una struttura di immagine del processo tipica di questa strategia: uno stack del kernel separato è utilizzato per gestire chiamate e ritorni, mentre il processo è in modalità kernel, mentre il codice e i dati del sistema operativo sono nello spazio degli indirizzi condiviso e sono comuni a tutti i processi utente.

Quando si verificano un'interruzione, una trap o una chiamata a supervisore, il processore è posto in modalità kernel e il controllo passa al sistema operativo. A questo fine è salvato il

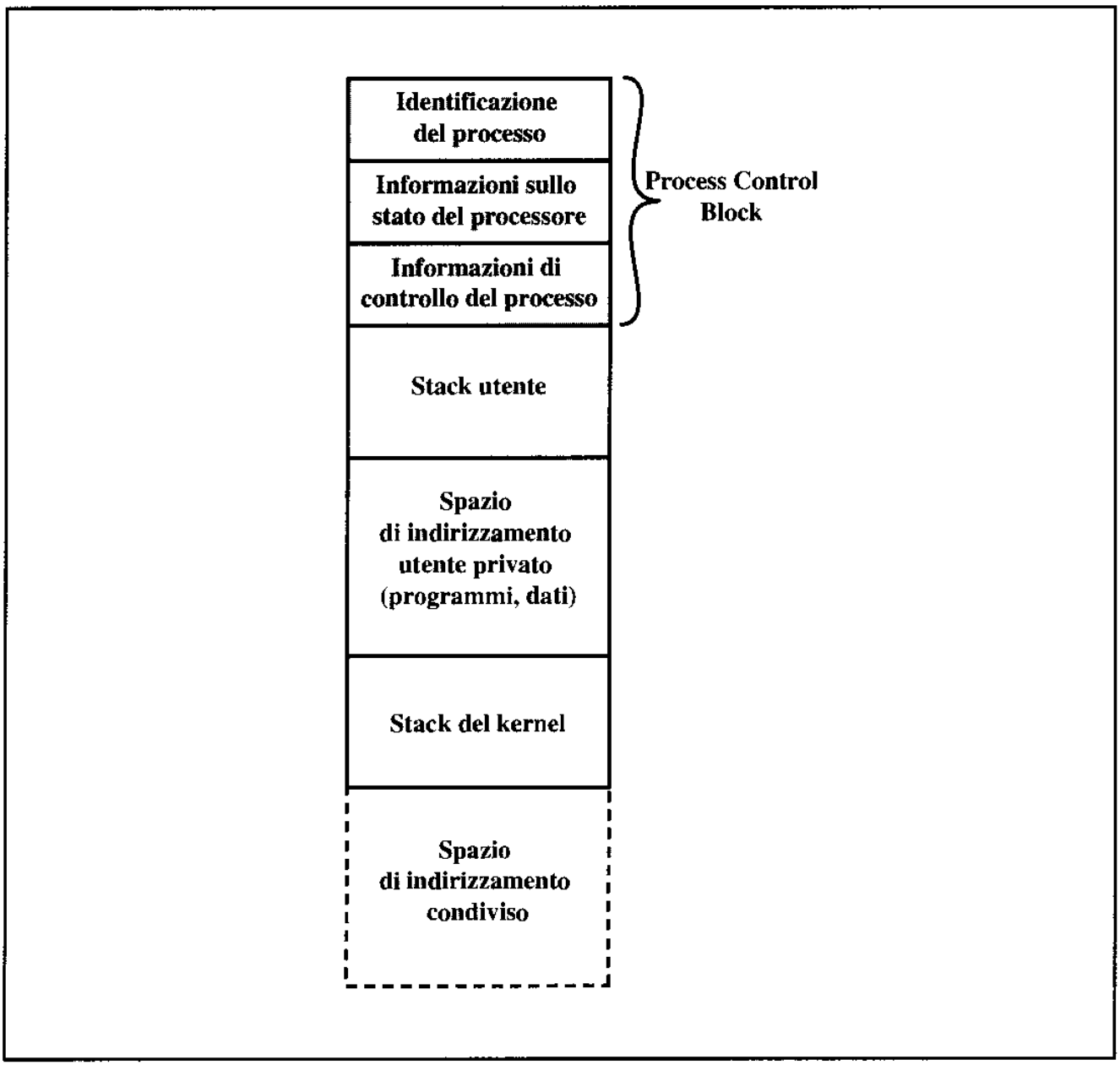


Figura 3.14 *Immagine del processo: il sistema operativo viene eseguito all'interno dello spazio utente*

contesto di modo, e un cambio di modo fa posto a una routine del sistema operativo, ma l'esecuzione continua entro il processo corrente. Pertanto, non è effettuato un cambio di processo, ma solo un cambio di modo entro lo stesso processo: se il sistema operativo, completato il suo lavoro, stabilisce che il processo corrente dovrebbe continuare l'esecuzione, allora un cambio di modo ripristina il programma interrotto. Questo è uno dei vantaggi fondamentali di questo approccio: un programma utente è stato interrotto per utilizzare una routine del sistema operativo e quindi ripristinato, senza dover ricorrere a due cambi di processo. Se comunque si stabilisce che debba avvenire un cambio di processo per ritornare al programma precedentemente in esecuzione, allora il controllo passa ad una routine per il cambi dei processi. A seconda dell'architettura del sistema, questa routine può essere eseguita nel processo corrente, oppure no: in ogni

caso, ad un certo punto, il processo corrente deve essere posto in uno stato di non-esecuzione, e un altro sarà scelto come processo in esecuzione. Durante questa fase è logicamente molto più conveniente considerare che l'esecuzione avvenga al di fuori di tutti i processi.

Questa visione del sistema operativo è, in un certo senso, piuttosto insolita: in parole povere, potremmo dire che, in determinati momenti, un processo salverà le proprie informazioni di stato, fra quelli pronti ne sceglierà uno per l'esecuzione e gli cederà il controllo. Durante il tempo critico, il codice eseguito nel processo utente è codice condiviso di sistema operativo, e non codice utente, ed è per questo che la situazione non è arbitraria o caotica. Per come sono concepite le modalità utente e kernel, l'utente non può interferire con le routine del sistema operativo o danneggiarle, anche se sono in esecuzione nel suo ambiente di processo. Questo ci ricorda inoltre che esiste una distinzione fra i concetti di processo e programma, e che la relazione fra i due non è uno a uno. Entro un processo possono essere eseguiti sia un programma utente sia programmi del sistema operativo, e questi, anche se sono eseguiti all'interno di diversi processi utente, sono identici.

Sistemi operativi basati sui processi

Un'ulteriore possibilità (Figura 3.13c) consiste nell'implementare il sistema operativo come un insieme di processi di sistema. Come nelle altre opzioni, il software che è parte del kernel, è eseguito in modalità kernel; in questo caso in ogni modo le principali funzioni del kernel sono organizzate come processi separati. Di nuovo, ci può essere una piccola quantità di codice per lo scambio, eseguito al di fuori di ogni processo.

Quest'approccio ha diversi vantaggi: impone una disciplina nella progettazione dei programmi, che incoraggia l'uso di un sistema operativo modulare con interfacce minimali e pulite fra i moduli. Inoltre, alcune funzioni non critiche sono implementate convenientemente come processi separati. Ad esempio, avevamo già ricordato il caso di un programma che registra il livello di utilizzo di diverse risorse (processore, memoria, canali) e il ritmo di avanzamento dei processi utente nel sistema. Poiché questo programma non fornisce un servizio particolare ad alcun processo attivo, può essere chiamato solo dal sistema operativo. Come processo, la funzione può essere eseguita ad un livello assegnato di priorità, e può essere interallacciata con altri processi, sotto il controllo di un allocatore. Infine, l'implementazione del sistema operativo come un insieme di processi è utile in un ambiente multiprocessore o multicomputer, in cui alcuni dei servizi del sistema operativo possono essere affidati a processori dedicati, migliorando le prestazioni.

3.4 Gestione dei processi in UNIX SVR4

UNIX System V usa un semplice, ma potente meccanismo di gestione dei processi, ben visibile all'utente. UNIX segue il modello della Figura 13.13b, in cui la maggior parte del sistema operativo è eseguito nell'ambiente di un processo utente, richiedendo quindi due modalità, utente e kernel. UNIX usa due categorie di processi: di sistema e utente. I processi di sistema

eseguono codice di sistema operativo in modalità kernel per eseguire funzioni amministrative e di gestione, come l'allocazione della memoria e il trasferimento dei processi su disco. I processi utente operano in modalità utente per eseguire programmi e funzioni di utilità, e in modalità kernel, per eseguire istruzioni appartenenti al kernel. Un processo utente entra in modalità kernel mediante una chiamata di sistema, quando è generata un'eccezione, o quando avviene un'interruzione.

Stati dei processi

Il sistema UNIX riconosce un totale di nove stati di processo, elencati nella Tabella 3.10; la Figura 3.15 (ripresa da una figura in [BACH86]) presenta un diagramma delle transizioni di stato, simile a quello della Figura 3.6, con i due stati "dormienti" che corrispondono ai due stati Blocked. Le differenze si possono riassumere in breve:

- UNIX impiega due stati Running per indicare se il processo è eseguito in modalità utente oppure kernel.
- È effettuata una distinzione fra i due stati Ready in Memory e Preempted (prerilasciato). Si tratta essenzialmente dello stesso stato, come segnala la linea tratteggiata che li unisce, ma la distinzione è necessaria per sottolineare il modo in cui un processo passa nello stato Preempted. Quando un processo è eseguito in modalità kernel (in seguito ad una chiamata a supervisore, un'interruzione di clock o di I/O), verrà il momento in cui il kernel ha completato il proprio lavoro, ed è pronto a restituire il controllo al programma utente. A questo punto il kernel può decidere di prerilasciare il processo corrente a favore di uno pronto e dotato di più alta priorità. In questo caso il processo corrente passa allo stato Preempted. Comunque, per facilitare

Tabella 3.10 *Stati dei processi in UNIX*

User Running	Esecuzione in modalità utente
Kernel Running	Esecuzione in modalità kernel
Ready, in Memory	Pronto per l'esecuzione appena il kernel lo schedula
Asleep, in Memory	Impossibilitato ad eseguire finchè non avviene un evento; il processo è in memoria principale.
Ready, swapped	Il processo è pronto per l'esecuzione, ma deve essere caricato in memoria principale prima che il kernel lo possa schedulare per l'esecuzione.
Sleeping, swapped	Il processo sta attendendo un evento ed è stato scaricato su un dispositivo di memoria secondaria
Preempted	Il processo sta ritornando dalla modalità kernel a quella utente, ma il kernel lo prerilascia e passa a schedulare un altro processo.
Created	Il processo è stato creato da poco e non è ancora pronto per l'esecuzione
Zombie	Il processo non esiste più, ma rimangono informazioni relative al processo utilizzate dal genitore

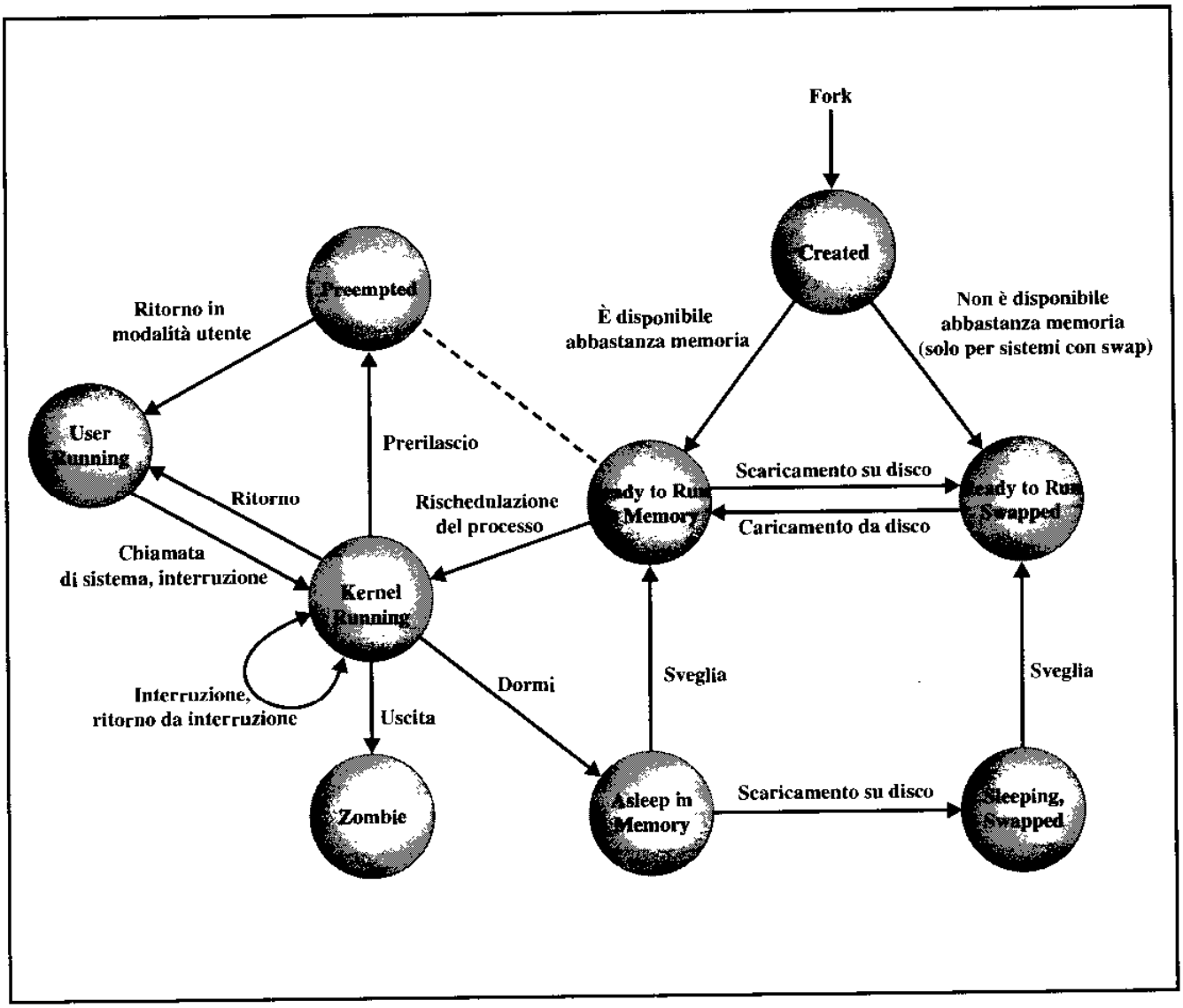


Figura 3.15 *Diagramma delle transizioni di stato dei processi in UNIX*

l'allocazione, i processi in stato Preempted e quelli Ready in Memory formano un'unica coda.

La prelazione può avvenire solo quando un processo sta per passare dalla modalità kernel a quella utente. Quando un processo è in esecuzione in modalità kernel non può essere prerilasciato, e ciò rende UNIX inutilizzabile per l'elaborazione in tempo reale (di cui tratterà il Capitolo 10).

Due processi sono unici in UNIX. Il processo 0 è un processo speciale creato quando parte il sistema; in effetti, è una struttura dati predefinita, caricata al boot del sistema: è il processo che gestisce il trasferimento su disco. Inoltre il processo 0 genera il processo 1, chiamato *init;* tutti gli altri processi del sistema, hanno 1 come antenato; quando un nuovo utente interattivo entra nel sistema, init crea per lui un processo utente. In seguito, il processo utente può creare un albero di processi figli, in modo tale che ciascuna particolare applicazione consista di diversi processi correlati.

Descrizione dei processi

Un processo in UNIX è un insieme di strutture piuttosto complesso che fornisce al sistema operativo tutte le informazioni necessarie per gestire i processi. La Tabella 3.11 riassume gli elementi dell'immagine del processo, organizzati in tre contesti: a livello utente, dei registri, a livello sistema.

Il **contesto a livello utente** contiene gli elementi fondamentali di un programma utente, e può essere generato direttamente da un file oggetto compilato. Il programma utente è suddiviso nelle aree testo e dati; l'area testo è a sola lettura e serve a memorizzare le istruzioni del programma. Mentre il processo è in esecuzione, il processore usa l'area dello stack utente, per le chiamate e i ritorni da procedura, e il passaggio dei parametri. L'area della memoria condivisa è

Tabella 3.11 *Immagine del processo in UNIX*

	Contesto a livello utente
Testo del processo	Istruzioni macchina eseguibili del programma
Dati del processo	Dati cui il processo può accedere
Stack utente	Contiene gli argomenti, le variabili locali e i puntatori per le funzioni eseguite in modalità utente
Memoria condivisa	Memoria condivisa con altri processi, utilizzata per la comunicazione fra i processi
	Contesto dei registri
Program counter	Indirizzo dell'istruzione da eseguire successivamente; può essere nella memoria kernel o utente del processo.
Registro di stato del processore	Contiene lo stato dell'hardware al momento del prerilascio; i contenuti e il formato dipendono dall'hardware
Puntatore allo stack	Punta alla cima dello stack kernel o utente, in dipendenza dal modo operativo al momento del prerilascio.
Registri di tipo generale	Dipendono dall'hardware
	Contesto a livello di sistema
Entry nella tabella dei processi	Definisce lo stato di un processo; questa informazione è sempre accessibile al sistema operativo.
U Area (area utente)	Informazioni di controllo del processo cui è necessario accedere solo nel contesto del processo.
Tabella delle regioni per processo	Definisce la mappatura dagli indirizzi virtuali a quelli fisici; contiene anche un campo dei permessi che indica il tipo degli accessi consentiti al processo: sola lettura, lettura-scrittura o lettura-esecuzione.
Stack del kernel	Contiene il frame dello stack delle procedure del kernel, quando il processo viene eseguito in modalità kernel.

un'area dati in comune con gli altri processi: esiste solo una copia fisica di un'area condivisa di memoria, ma, tramite la memoria virtuale, appare a ciascun processo come appartenente al suo spazio d'indirizzamento. Quando un processo non è in esecuzione, le informazioni di stato del processore sono memorizzate nell'area riservata al **contesto dei registri**.

Il contesto a livello di sistema contiene le informazioni rimanenti di cui il sistema operativo ha bisogno per gestire i processi: si tratta di una parte statica, di dimensioni fisse, che permane lungo tutto il ciclo vitale di un processo, e di una parte dinamica, che ha invece dimensioni variabili. Un elemento della parte statica è l'entry della tabella dei processi, che fa effettivamente parte della tabella dei processi, mantenuta dal sistema operativo con un'entry per processo. L'entry della tabella dei processi contiene informazioni per il controllo del processo sempre accessibili al kernel, perciò, in un sistema a memoria virtuale tutte le entry della tabella dei processi sono mantenute in memoria principale. La Tabella 3.12 elenca i contenuti di un'entry della tabella dei processi. L'area utente, o U area, contiene informazioni aggiuntive per il con-

Tabella 3.12 *Entry della Tabella dei processi di UNIX*

Stato del processo	Stato corrente del processo
Puntatori	Alla U Area e all'area di memoria del processo (testo, dati, stack)
Dimensione del processo	Permette al sistema operativo di sapere quanto spazio allocare per il processo
Identificatori dell'utente	L'**ID utente reale** identifica l'utente responsabile del processo in esecuzione. L'**ID utente effettivo** può essere usato da un processo per ottenere privilegi temporanei associati ad un particolare programma; mentre quel programma viene eseguito come parte del processo, il processo opera con l'ID utente effettivo.
Identificatori dei processi	ID del processo e del processo genitore. Sono impostati quando il processo viene creato durante la chiamata di sistema fork.
Descrittore degli eventi	Valido quando un processo è in uno stato dormiente; quando avviene l'evento il processo passa allo stato Ready.
Priorità	Usata per la schedulazione del processo.
Segnali	Elenca i segnali mandati al processo ma non ancora gestiti.
Timer	Includono il tempo di esecuzione del processo, l'uso delle risorse kernel e il timer impostato dall'utente utilizzato per mandare un segnale di allarme a un processo.
P_link	Puntatore al link successivo nella coda dei Ready (valido se il processo è nello stato di Ready).
Stato della memoria	Indica se l'immagine del processo è in memoria principale o scaricata sul disco. Se è in memoria questo campo indica anche se può essere scaricato o temporaneamente bloccato (locked) nella memoria principale.

trollo dei processi di cui il kernel ha bisogno quando è eseguito nel contesto del processo stesso; è utilizzata anche per la paginazione dei processi da e verso la memoria. La Tabella 3.13 presenta i contenuti di questa tabella.

La distinzione fra l'entry della tabella dei processi e la U area riflette il fatto che il kernel UNIX è sempre eseguito nel contesto di qualche processo. Nella maggior parte dei casi, il kernel avrà a che fare con ciò che riguarda quel processo; talvolta però, quando ad esempio il kernel sta eseguendo un algoritmo di schedulazione, dovrà accedere alle informazioni relative agli altri processi.

La terza parte statica del contesto a livello di sistema è la tabella delle regioni per processo, utilizzata dal sistema per la gestione della memoria. Infine, lo stack del kernel è la porzione dinamica del contesto a livello di sistema: esso è usato quando il processo è eseguito in modalità kernel, e contiene le informazioni che devono essere salvate e ripristinate quando avviene una chiamata di procedura o una interruzione.

Tabella 3.13 *U Area in UNIX*

Puntatore alla tabella dei processi	Indica l'entry che corrisponde all'U area
Identificatori dell'utente	ID utente reale ed effettivo. Utilizzato per determinare i privilegi dell'utente
Timer	Registrano il tempo di esecuzione in modalità utente e kernel del processo e dei suoi discendenti
Array per la gestione dei segnali	Per ciascun tipo di segnale definito nel sistema, indica come un processo reagirà alla ricezione di un determinato segnale (uscita, ignora, esegui una specificata funzione utente)
Terminale	Indica il terminale relativo al processo, se ne esiste uno
Campo di errore	Registra gli errori incontrati durante una chiamata di sistema
Valore di ritorno	Contiene il risultato di una chiamata di sistema
Parametri di I/O	Descrivono la quantità dei dati da trasferire, l'indirizzo dell'array dei dati sorgente (o di destinazione) nello spazio utente, e gli offset dei file per l'I/O.
Parametri dei file	La directory e la radice correnti descrivono l'ambiente dei file del processo
Tabella dei descrittori dei file utente	Registra i file aperti dal processo
Campi limite	Delimitano la dimensione del processo e dei file che può aprire
Campi per l'impostazione dei permessi	Mascherano l'impostazione dei modi di accesso ai file che il processo crea

Controllo dei processi

La creazione dei processi in UNIX è fatta mediante una chiamata di sistema al kernel detta fork (). Quando un processo esegue una richiesta di fork, il sistema operativo esegue le seguenti funzioni ([BACH86]):

1. Alloca uno spazio nella tabella dei processi per il nuovo processo
2. Assegna un ID di processo al figlio, unico nel sistema.
3. Fa una copia dell'immagine del processo genitore, ad eccezione della memoria condivisa.
4. Incrementa i contatori dei file del genitore per registrare che ora anche il nuovo processo possiede questi file.
5. Assegna al processo figlio lo stato Ready.
6. Ritorna l'identificatore ID del figlio al genitore, e il valore 0 al figlio.

Tutto questo lavoro è svolto in modalità kernel nel processo genitore; quando il kernel ha compiuto queste funzioni, può fare una delle cose seguenti, come parte della routine dell'allocatore:

1. Rimanere nel processo genitore: il controllo ritorna al modo utente nel punto di chiamata della fork del genitore.
2. Trasferire il controllo al processo figlio: il processo figlio inizia ad eseguire dallo stesso punto del codice del genitore, cioè dal ritorno della chiamata fork.
3. Trasferire il controllo ad un altro processo: sia il genitore sia il figlio sono lasciati nello stato Ready.

È piuttosto difficile visualizzare questo metodo di creazione dei processi, perché sia il genitore sia il figlio stanno eseguendo la stessa porzione di codice; la differenza si scopre controllando il valore di ritorno dalla routine fork: se è zero, allora è in esecuzione il processo figlio, che esegue un salto all'opportuno programma utente per continuare l'esecuzione. Se il valore non è zero, allora è in esecuzione il genitore, e la linea principale di esecuzione può continuare.

3.5 Sommario

Il principale blocco costruttivo di un sistema operativo moderno è il processo, per cui la funzione fondamentale del sistema operativo è creare, gestire e terminare i processi. Mentre i processi sono attivi, il sistema controlla che a ciascuno sia assegnato tempo di processore per l'esecuzione, coordina le loro attività, gestisce le richieste conflittuali e alloca risorse di sistema per i processi.

Per gestire i processi il sistema mantiene una descrizione, o immagine, di ciascun processo, che contiene lo spazio degli indirizzi entro cui il processo è eseguito e un process control block. Quest'ultimo contiene tutte le informazioni richieste dal sistema operativo per gestire il processo, fra cui il suo stato corrente, le ricorse che gli sono allocate, la priorità e altri dati rilevanti.

Durante il proprio ciclo di vita, un processo attraversa diversi stati, i più importanti dei quali sono Ready, Running e Blocked. Un processo Ready non è correntemente in esecuzione, ma può esserlo, appena sarà scelto dal sistema operativo. Il processo Running è quello correntemente eseguito dal processore; in un sistema multiprocessore può trovarsi in questo stato più di un processo. Un processo Blocked è in attesa del completamento di qualche evento, ad esempio un'operazione di I/O.

Un processo in esecuzione è interrotto da un'interruzione, che è un evento esterno al processo riconosciuto dal processore, oppure dall'esecuzione di una chiamata a supervisore al sistema operativo.

In entrambi i casi il processore effettua un cambio di modo, trasferendo il controllo a una routine del sistema operativo. Il sistema operativo, completato il lavoro necessario, può riprendere il processo interrotto o può passare ad un altro.

3.6 Letture raccomandate

Tutti i libri elencati nella Sezione 2.9 riguardano i contenuti di questo capitolo. La gestione dei processi UNIX è ben descritta in [GOOD94] e [GRAY97]. [NEHM75] propone un'interessante esame sugli stati dei processi e sulle primitive necessarie per l'allocazione.

GOOD94 Goodheart, B., e Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*, Englewood Cliffs, NJ: Prentice Hall, 1994.

GRAY97 Gray, J. *Interprocess Communication in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

NEHM75 Nehmer, J. "Dispatcher Primitives for the Construction of Operating System Kernels" *Acta Informatica*, Vol. 5 , 1975.

3.7 Problemi

3.1 Elenca cinque attività principali di un sistema operativo inerenti la gestione dei processi, e descrivi brevemente perché ciascuna di esse è indispensabile.

3.2 In [PINK89] sono definiti i seguenti stati dei processi: Running, Ready (Attivo), Blocked, Suspend. Un processo è bloccato se è in attesa di utilizzare una risorsa, ed è sospeso se attende il completamento di un'operazione relativa ad una risorsa che ha già acquisito. In

molti sistemi operativi questi due stati sono considerati come un unico stato Blocked, mentre per Suspend vale la descrizione fornita in questo capitolo. Confrontare i rispettivi meriti dei due insiemi di definizioni.

3.3 Per il modello a sette stati della Figura 3.7b, disegnare un diagramma di code simile a quello della Figura 3.6b.

3.4 Si consideri il diagramma delle transizioni di stato della Figura 3.7b. Si supponga di essere nel momento in cui il sistema operativo effettua l'allocazione di un processo, e sono presenti processi sia nello stato Ready, che in quello Ready-Suspend e almeno un processo nello stato Ready-Suspend ha una priorità più alta di quella dei processi pronti. Due politiche opposte sono le seguenti: (1) scegliere sempre un processo Ready, per minimizzare i trasferimenti su disco, e (2) dare la preferenza al processo con priorità più alta anche se questo può significare un trasferimento su disco non strettamente necessario. Suggerire una politica intermedia che tenti di bilanciare il rispetto delle priorità e le prestazioni.

3.5 La Tabella 3.14 mostra gli stati dei processi del sistema operativo VAX/VMS.

a. Puoi fornire una giustificazione per l'esistenza di così tanti stati di attesa distinti?

b. Perché gli stati seguenti non hanno versioni residenti e trasferite su disco: attesa per fault di pagina, attesa per collisione di pagina, attesa per un evento comune, attesa per una pagina libera e attesa per una risorsa?

c. Disegnare il diagramma delle transizioni di stato ed indicare l'azione o l'evento che causa ciascuna transizione.

3.6 Il sistema operativo VAX/VMS utilizza quattro modalità d'accesso al processore per facilitare la protezione e la condivisione delle risorse di sistema fra i processi. Il modo d'accesso determina:

- **I privilegi di esecuzione delle istruzioni**: quali istruzioni il processore può eseguire.
- **I privilegi di accesso alla memoria**: a quali locazioni della memoria virtuale può accedere l'istruzione corrente.

I quattro modi sono i seguenti:

- **Kernel**: esegue il kernel del sistema operativo VMS, compresa la gestione della memoria e delle interruzioni e le operazioni di I/O.
- **Executive**: esegue molte delle chiamate di servizio al sistema operativo, tra cui le routine di gestione dei file e dei record (disco e nastro).
- **Supervisore**: esegue altri servizi del sistema operativo, come le risposte ai comandi utente.
- **Utente**: esegue i programmi utente, oltre a funzioni di utilità (compilatori, editor, linker e debugger).

Tabella 3.14 *Stati dei processi VAX/VMS*

Stati del processo	Condizione del processo
In esecuzione	Processo in esecuzione
Eseguibile (residente)	Pronto e residente in memoria centrale
Eseguibile (su disco)	Pronto ma scaricato dalla memoria centrale
In attesa per fault di pagina	Il processo ha fatto riferimento ad una pagina che non è in memoria centrale, e deve aspettare che la pagina venga letta.
In attesa per collisione di pagina	Il processo ha fatto riferimento ad una pagina condivisa, di cui è già in attesa un altro processo per un precedente fault di pagina, oppure ad una pagina privata che si sta leggendo o scrivendo
Attesa di un evento comune	In attesa del flag di un evento condiviso (i flag degli eventi sono meccanismi per l'invio di segnali di un bit fra processi)
In attesa di una pagina libera	In attesa che una pagina libera in memoria centrale sia aggiunta all'insieme delle pagine dedicate al processo (insieme di lavoro o working set del processo)
In attesa ibernato (residente)	È il processo stesso a mettersi in stato di attesa
In attesa ibernato (su disco)	Il processo ibernato è scaricato dalla memoria principale
In attesa di evento locale (residente)	Il processo è in memoria principale ed in attesa di un flag di evento locale (di solito il completamento di un'operazione di I/O)
In attesa di evento locale (su disco)	Il processo in attesa di un evento locale è scaricato dalla memoria principale
In attesa sospeso (residente)	Il processo è posto in stato di attesa da un altro processo
In attesa sospeso (su disco)	Il processo sospeso è scaricato dalla memoria principale
In attesa di risorsa	Il processo è in attesa di una risorsa di sistema

Un processo in esecuzione in un modo meno privilegiato spesso deve chiamare una procedura eseguita in un modo più privilegiato, ad esempio quando un programma utente richiede un servizio del sistema operativo. Questa chiamata è ottenuta attraverso un'istruzione di cambio di modo (CHM), che genera un'interruzione, trasferendo il controllo ad una routine del nuovo modo di accesso. Il ritorno è effettuato eseguendo l'istruzione REI (Ritorno da eccezione o interruzione).

a. Diversi sistemi operativi hanno due modalità: kernel e utente. Quali sono i vantaggi e gli svantaggi nel disporre di quattro modi invece che di due?

b. Riesci a perorare la causa dei sistemi operativi che dispongono di più di quattro modi?

3.7 Lo schema VMS presentato nel problema precedente spesso è definito come struttura di protezione ad anello (Figura 3.16). Il semplice schema kernel/utente, descritto nella Sezione 3.3 è una struttura a due anelli. [SILB94] evidenzia un problema connesso a quest'approccio;

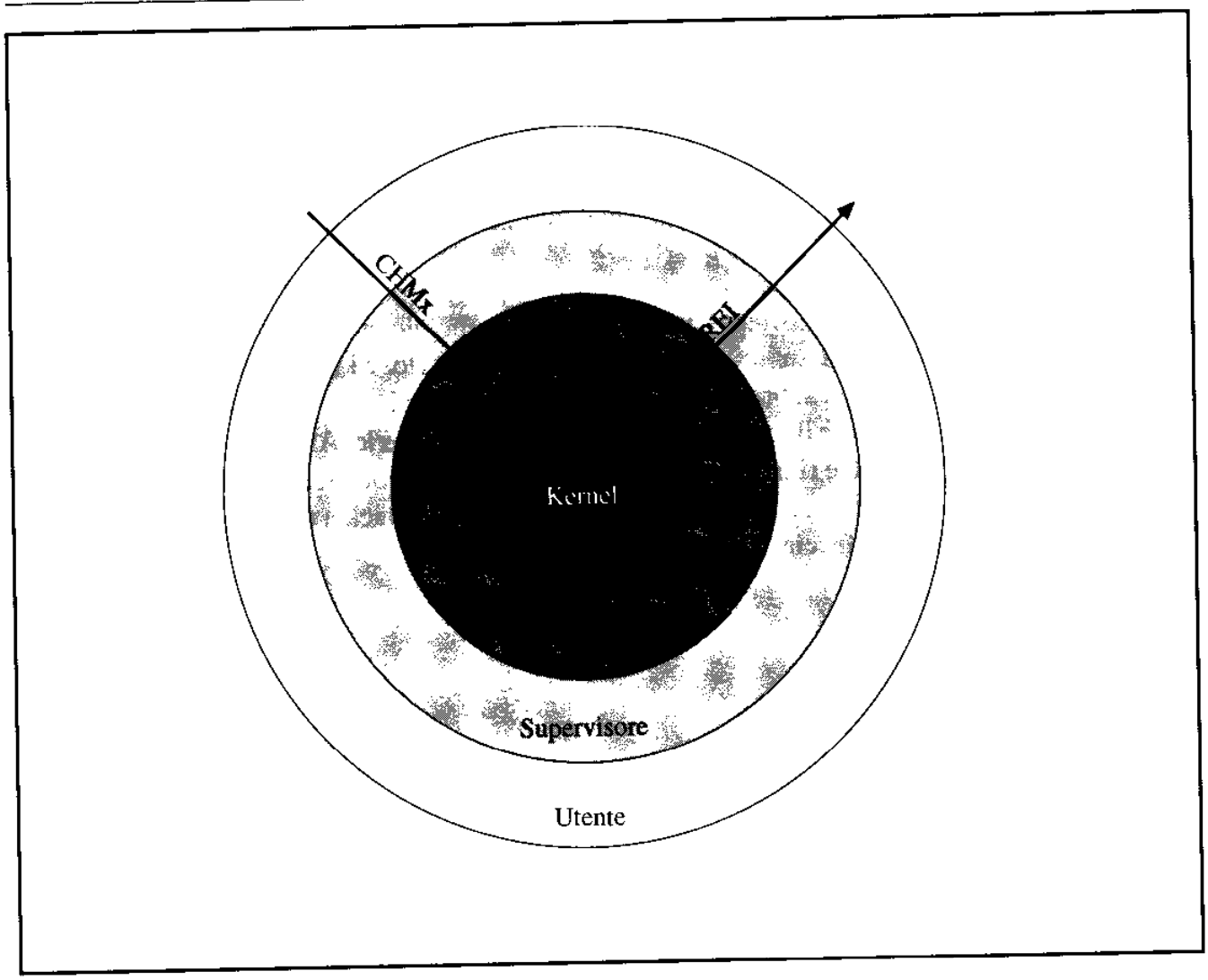


Figura 3.16 *Modi di accesso in VAX/VMS*

"Il principale svantaggio della struttura ad anello (gerarchica) è che non permette di applicare il principio della necessità-di-conoscere. In particolare, se un oggetto deve essere accessibile nel dominio D_j, ma non lo è nel dominio D_i, allora dobbiamo avere $j < i$. Ma questo significa che ogni segmento accessibile in D_i è anche accessibile in D_j."

a. Spiega chiaramente in cosa consiste il problema presentato nella precedente citazione.

b. Suggerisci un modo tramite il quale un sistema operativo strutturato ad anello possa affrontare questo problema.

3.8 La Figura 3.6b suggerisce che un processo può essere solo in una coda di eventi alla volta.

a. È possibile permettere ad un processo di attendere più di un evento alla volta? Fornisci un esempio.

b. In caso affermativo, come si potrebbe modificare la struttura di code della figura per supportare questa nuova caratteristica?

3.9 In alcuni dei primi computer, un'interruzione provocava la memorizzazione dei valori dei registri in locazioni fisse, associate ad un determinato segnale di interruzione. In quali circostanze questa è una tecnica pratica? Spiegare perché non è in generale conveniente.

3.10 Si era affermato nella Sezione 3.5 che UNIX non è utilizzabile per applicazioni in tempo reale, perché un processo eseguito in modalità kernel non può essere prerilasciato. Approfondire la questione.

3.11 Il kernel UNIX secondo le necessità farà crescere dinamicamente lo stack di un processo in memoria virtuale, ma non cercherà mai di restringerlo. Si consideri il caso in cui un programma chiami una subroutine C che allochi un array locale sullo stack, che consumi 10K. Il kernel espanderà il segmento dello stack per accoglierlo. Al ritorno dalla subroutine, il puntatore allo stack è ripristinato, e questo spazio potrebbe essere rilasciato dal kernel, ma non lo è. Spiegare perché sarebbe possibile ridurre lo stack a questo punto, e perché il kernel UNIX non lo faccia.

CAPITOLO 4

THREAD, SMP E MICROKERNEL

In questo capitolo si esaminano i concetti più avanzati di gestione dei processi, presenti in molti sistemi operativi di oggi. Per prima cosa, si mostra che il concetto di processo è più complesso di quanto visto finora, e incorpora due concetti distinti e indipendenti: possesso delle risorse ed esecuzione. Tale distinzione ha portato allo sviluppo, in alcuni sistemi operativi, di un costrutto chiamato *thread*. Dopo aver esaminato i thread, si considera SMP (*symmetric multiprocessing*, esecuzione simmetrica di più processi), in cui il sistema operativo deve schedulare simultaneamente diversi processi su vari processori. Infine si introduce il concetto di microkernel (micronucleo), un modo efficace di strutturare il sistema operativo per supportare la gestione dei processi e le altre attività.

4.1 Processi e thread

Finora si è visto che un processo è contemporaneamente:

- **Un elemento che possiede delle risorse**: un processo possiede uno spazio di indirizzamento virtuale che contiene l'immagine del processo, e saltuariamente può richiedere ulteriore memoria e il controllo di altre risorse, quali canali di I/O, dispositivi e file.
- **Un elemento che viene allocato**: un processo è una traccia di esecuzione entro uno o più programmi. Tale esecuzione può essere alternata con quella di altri processi, quindi un processo ha uno stato di esecuzione (Running, Ready, ecc.) e una priorità di schedulazione. Il sistema operativo ha il compito di schedulare e allocare i processi.

In molti sistemi operativi queste due caratteristiche costituiscono l'essenza di un processo. In ogni modo, riflettendoci, il lettore dovrebbe convincersi che queste due caratteristiche sono indipendenti, e potrebbero essere gestite separatamente dal sistema (cosa che fanno molti sistemi operativi tra i più recenti). Per distinguerle, l'elemento che viene allocato viene detto **thread**, o **lightweight process** (processo leggero), mentre l'elemento che possiede le risorse è chiamato **processo**, o **task**[1].

Multithreading

Il multithreading (*thread multipli*) è la capacità di un sistema operativo di supportare thread di esecuzione multipli per ogni processo; l'approccio tradizionale di un singolo thread di esecuzione per ogni processo, in cui il concetto di thread non è evidenziato, è definito approccio a thread singolo. I due riquadri di sinistra della Figura 4.1 mostrano approcci a thread singolo. MS-DOS è un esempio di sistema operativo che supporta un singolo processo utente con un solo thread; altri sistemi operativi, come UNIX, supportano processi multipli composti di un solo thread ciascuno. La parte destra della Figura 4.1 mostra approcci con thread multipli. Un motore runtime di Java è un esempio di sistema con un processo composto da thread multipli. In questa sezione si esamina l'uso di processi multipli, ognuno dei quali supporta thread multipli; tale approccio è adottato, fra gli altri, da Windows NT, Solaris, Mach e OS/2. In questa sezione viene data una descrizione generale del multithreading; i dettagli di Windows NT e Solaris sono discussi nelle sezioni 4.4 e 4.5.

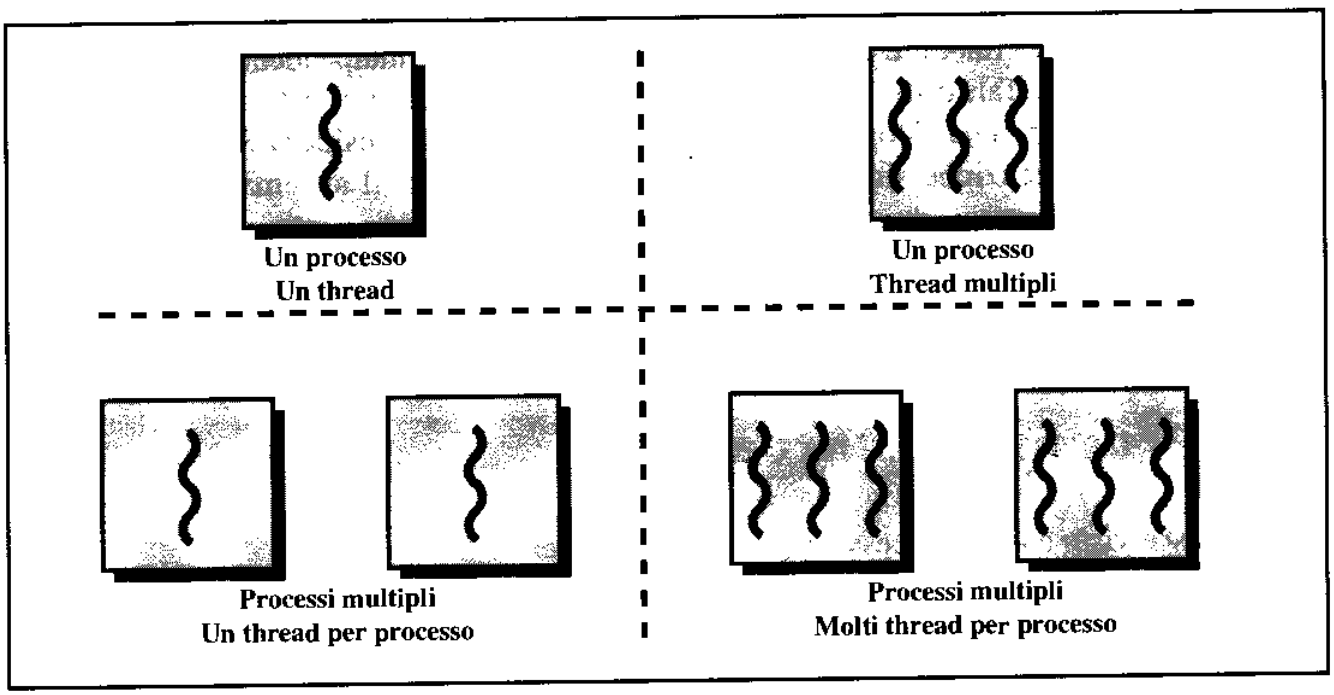


Figura 4.1 *Thread e processi [ANDE97]*

[1] Comunque, anche questo grado di coerenza non viene mantenuto. In OS/390, i concetti di spazio di indirizzamento e task, rispettivamente, corrispondono a grandi linee ai concetti di processo e thread qui descritti. Inoltre, il [illegible] *lightweight process* è usato come: (1) equivalente del termine *thread*; (2) tipo particolare di thread noto come thread a livello di kernel; (3) nel caso di Solaris, un'entità che mappa thread a livello utente in thread a livello di kernel.

In un ambiente con thread multipli, un processo è definito come l'elemento di protezione e l'elemento di allocazione delle risorse. Ad ogni processo sono associati:

- Uno spazio di indirizzamento virtuale che contiene l'immagine del processo.
- Accesso protetto ai processori, ad altri processi (per comunicazione fra processi), file e risorse di I/O (dispositivi e canali).

All'interno di un processo ci possono essere uno o più thread, ognuno avente:

- Uno stato di esecuzione del thread (Running, Ready, ecc.).
- Un contesto, che è salvato quando il thread non è in esecuzione; un thread può essere visto come un program counter indipendente relativo ad un processo.
- Uno stack di esecuzione.
- Uno spazio di memoria statico per le variabili locali del thread.
- Accesso alla memoria e alle risorse del processo di cui fa parte, condiviso con gli altri thread componenti il processo.

La Figura 4.2 illustra la distinzione fra thread e processi, dal punto di vista della gestione dei processi. In un modello a thread singolo (privo del concetto di thread), la rappresentazione di un processo contiene il suo Process Control Block e il suo spazio di indirizzamento utente, nonché

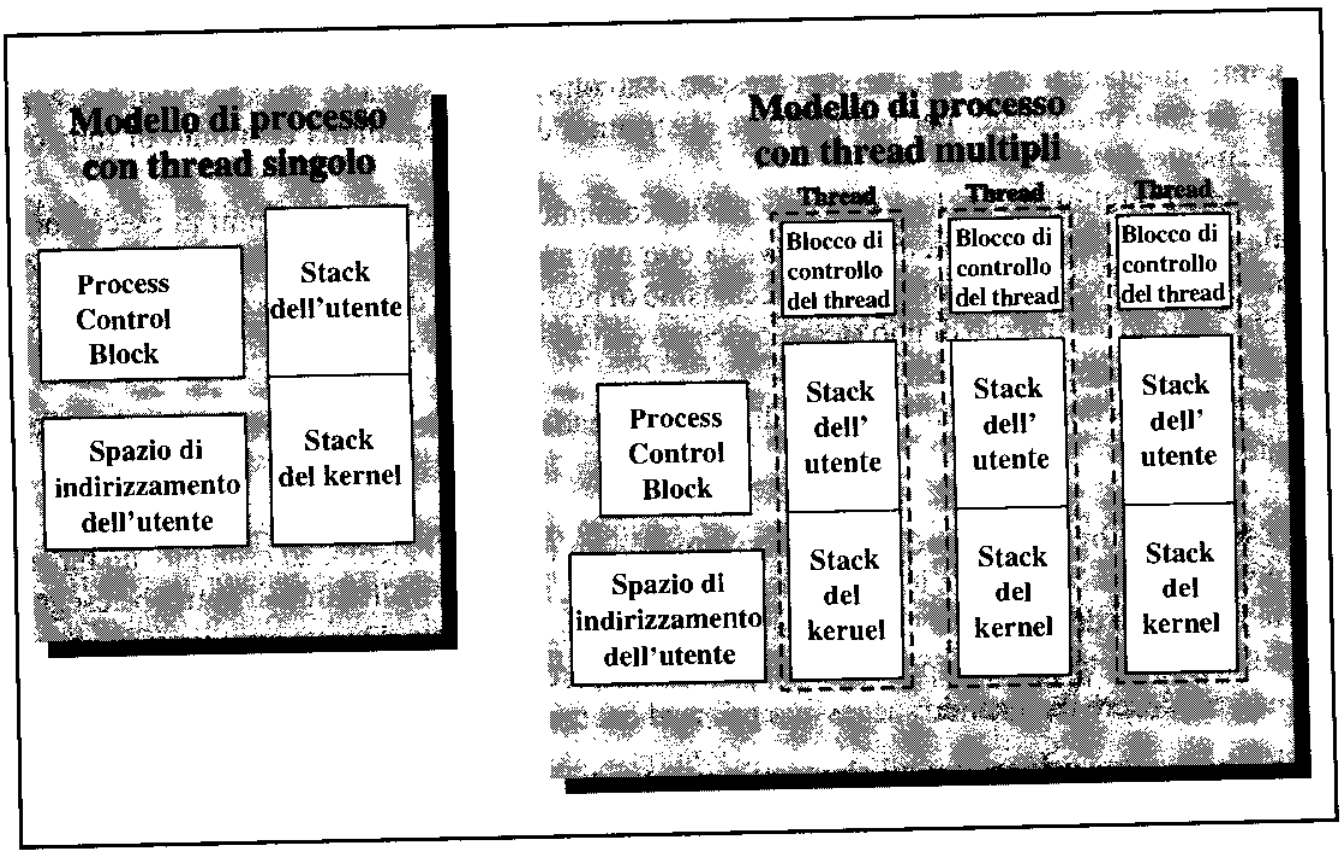


Figura 4.2 *Modelli di processi con thread singolo e multiplo*

lo stack utente e lo stack del kernel, per controllare il comportamento delle chiamate di procedura e i ritorni da chiamata durante l'esecuzione del processo. Mentre il processo è in esecuzione, esso controlla i registri del processore, e il contenuto dei registri viene salvato quando il processo non è in esecuzione. In un ambiente multithreading, ogni processo ha associato un solo Process Control Block e un solo spazio di indirizzamento utente, ma ogni thread ha un proprio stack, un blocco di controllo privato contenente l'immagine dei registi, la priorità e altre informazioni relative al thread.

Quindi, tutti i thread componenti un processo condividono lo stato e le risorse di quel processo, risiedono nello stesso spazio di indirizzamento e hanno accesso agli stessi dati. Quando un thread altera un dato in memoria, gli altri thread vedono il risultato quando accedono a quel dato. Se un thread apre un file con privilegi di lettura, gli altri thread dello stesso processo possono leggere quel file.

Il vantaggio chiave dei thread è nelle prestazioni: la creazione di un nuovo thread in un processo esistente richiede molto meno tempo rispetto alla creazione di un processo nuovo, e lo stesso vale per la terminazione di un thread e il cambio tra due thread dello stesso processo. Quindi, se c'è un'applicazione o funzione da implementare come un insieme di unità di esecuzione correlate, è molto più efficiente utilizzare un insieme di thread rispetto ad un insieme di processi. Gli studi effettuati dai progettisti di Mach mostrano che l'aumento di efficienza nella creazione di processi rispetto ad un'implementazione di UNIX paragonabile, ma che non fa uso di thread, è di un fattore 10 [TEVA87].

Un esempio di applicazione che potrebbe far uso di thread è un server, come un file server su rete locale. Per ogni nuova richiesta di file, viene creato un nuovo thread per la gestione dei file. Siccome un server gestisce molte richieste, molti thread verranno creati e distrutti in breve tempo. Se il server è multiprocessore, molti thread appartenenti allo stesso processo possono essere eseguiti simultaneamente su diversi processori. Talvolta l'uso di thread è utile anche in presenza di un singolo processore, per semplificare la struttura di un programma costituito di funzioni logicamente distinte.

I thread migliorano anche l'efficienza della comunicazione fra i programmi in esecuzione. Nella maggior parte dei sistemi operativi, la comunicazione fra processi indipendenti richiede l'intervento del kernel per fornire un meccanismo di protezione e di comunicazione. Comunque, poiché i thread all'interno di uno stesso processo condividono memoria e file, possono comunicare fra loro senza chiamare il kernel.

[LETW88] contiene quattro esempi di uso dei thread in un sistema multiprocessing a singolo utente:

- **Esecuzione in foreground e in background**: in un foglio di calcolo, ad esempio, un thread potrebbe gestire i menu e leggere l'input dell'utente, mentre un altro thread potrebbe eseguire i comandi dell'utente e l'aggiornamento del foglio di calcolo. Questa strutturazione spesso migliora i tempi di risposta dell'applicazione percepiti dall'utente, poiché il programma può richiedere il comando successivo prima che il comando precedente sia stato completato.
- **Elaborazione asincrona:** gli elementi asincroni di un programma si possono implementare come thread. Ad esempio, come protezione contro interruzioni dell'erogazione di corrente elettrica, si può progettare un elaboratore di testo in modo che scriva su disco ogni minuto il

contenuto del buffer in memoria. Siccome si può creare un thread il cui unico compito sia il salvataggio periodico dei dati, e che si attivi automaticamente contattando il sistema operativo, non c'è bisogno di scrivere codice spurio nel programma principale per effettuare controlli periodici o coordinare l'input e l'output.

- **Velocità di esecuzione:** un processo multithread può calcolare parte dei dati e allo stesso tempo leggere nuovi dati da un dispositivo. In un sistema multiprocessing, thread multipli dello stesso processo possono essere eseguiti parallelamente.
- **Programmi di organizzazione:** programmi che comprendono varie attività, o un certo numero di sorgenti e destinazioni di input e output, sono più semplici da progettare e realizzare utilizzando i thread.

Schedulazione e allocazione vengono effettuati a livello di thread, quindi la maggior parte dell'informazione di stato riguardante l'esecuzione viene tenuta in strutture dati a livello di thread. Ci sono, comunque, molte azioni che influiscono su tutti i thread di un processo e che il sistema operativo deve gestire a livello di processo. La sospensione richiede la rimozione dello spazio di indirizzamento dalla memoria centrale e, poiché tutti i processi di un thread condividono lo stesso spazio di indirizzamento, tutti i thread devono entrare in stato di sospensione nello stesso tempo. Analogamente, la terminazione di un processo richiede la terminazione di tutti i thread che lo compongono.

Funzionalità dei thread

Come i processi, anche i thread hanno uno stato di esecuzione e possono sincronizzarsi fra loro. Questi due aspetti verranno esaminati uno alla volta.

Stato dei thread

Come per i processi, gli stati chiave di un thread sono Running, Ready e Blocked. In genere non ha senso associare stati Suspend ai thread, perché tali stati sono concettualmente operativi a livello di processo. In particolare, se un processo viene scaricato dalla memoria centrale, lo stesso deve accadere per tutti i suoi thread, perché condividono lo stesso spazio di indirizzamento.

Ci sono quattro operazioni di base associate ad un cambiamento dello stato di un thread:

- **Creazione:** tipicamente, quando un nuovo processo viene creato, si crea anche un thread. Successivamente, un thread di un processo può creare un altro thread per lo stesso processo, fornendo al nuovo thread il puntatore alle istruzioni e gli argomenti: vengono creati un contesto per i registri e gli stack, e il nuovo thread è messo nella coda Ready.
- **Blocco:** quando un thread deve aspettare un evento, entra in stato Blocked (salvando i registri utente, il program counter e lo stack pointer).
- **Sblocco:** all'occorrenza dell'evento su cui il thread era bloccato, il thread passa in stato Ready.

- **Terminazione:** quando un thread completa il suo compito, il suo contesto per i registri e i suoi stack vengono deallocati.

Un punto interessante è se il blocco di un thread provochi il blocco dell'intero processo. In altre parole, se un thread di un processo è bloccato, è possibile l'esecuzione di un altro thread dello stesso processo che si trova in stato Ready? Chiaramente, una parte della flessibilità e del potere dei thread si perderebbe se un thread bloccato causasse il blocco di un intero processo.

Questo punto verrà approfondito successivamente, nella trattazione di thread a livello di utente e thread a livello di kernel; per il momento si considerano i benefici, dal punto di vista delle prestazioni, dell'uso di thread che non bloccano l'intero processo. La Figura 4.3 (basata su quella in [KLEI96]) mostra un programma che effettua due chiamate a procedure remote (RPC, *remote procedure call*) a due diversi host per ottenere un risultato combinato. In un programma con un solo thread, i risultati sono ottenuti in sequenza, così il programma deve aspettare una risposta da ciascun server, uno dopo l'altro. Riscrivendo il programma, in modo da usare un thread separato per ogni RPC, otteniamo un sostanziale aumento di prestazioni. Si noti che, se questo programma opera su un solo processore, le richieste verranno generate sequenzialmente e i risultati ricevuti sequenzialmente; comunque, il programma può aspettare entrambe le risposte allo stesso tempo.

Su un sistema monoprocessore, la multiprogrammazione permette l'alternanza di thread multipli su processori multipli. Nell'esempio di Figura 4.4, tre thread appartenenti a due processi sono alternati sul processore. L'esecuzione passa da un thread ad un altro quando il thread corrente è bloccato, o quando la sua unità di tempo è esaurita[2].

Sincronizzazione dei thread

Tutti i thread di un processo condividono lo stesso spazio di indirizzamento e altre risorse, come i file aperti. Ogni alterazione di una risorsa effettuata da un thread modifica l'ambiente degli altri thread dello stesso processo; è quindi necessario sincronizzare le attività dei vari thread, così che essi non interferiscano l'uno con l'altro o rovinino le strutture dati. Per esempio, se due thread cercano entrambi di aggiungere un elemento ad una lista con doppio puntatore, un elemento può andare perso o la lista può diventare mal formata.

I problemi generati e le tecniche usate nella sincronizzazione di thread sono, in generale, analoghi al caso della sincronizzazione di processi; questo sarà argomento dei Capitoli 5 e 6.

Esempio: Aldus PageMaker

Un esempio d'uso dei thread è l'applicazione Aldus PageMaker sotto OS/2. PageMaker è un programma per la scrittura, progettazione e produzione per il desktop publishing. La struttura dei thread, mostrata in Figura 4.5 [KRON90], è stata scelta per ottimizzare i tempi di risposta dell'applicazione. Tre thread sono sempre attivi: un thread di gestione degli eventi, un thread per ridisegnare lo schermo e un thread di servizio.

---

[2] In questo esempio, il thread C inizia l'esecuzione dopo che il thread A ha esaurito la sua unità di tempo, anche se il thread B è pronto per l'esecuzione. La scelta fra B e C è una decisione dello scheduler, un argomento trattato nella Parte 4.

Generalmente, la gestione delle finestre in OS/2 rallenta se l'elaborazione di qualche messaggio in input richiede troppo tempo. OS/2 si basa sul criterio che nessun messaggio deve richiedere un tempo di elaborazione superiore a 0.1 secondi. Per esempio, chiamare un sottoprogramma per stampare una pagina durante l'elaborazione di un comando di stampa, impedirebbe al sistema di mandare messaggi alle altre applicazioni, rallentando le prestazioni. Per rispettare questo criterio, le operazioni complesse di PageMaker – stampa, lettura dei dati e disposizione del testo – sono effettuate dal thread di servizio, che si occupa anche di gran parte dell'inizializzazione del programma: in questo modo utilizza il tempo durante il quale l'utente chiama la finestra di dialogo per creare un nuovo documento o aprire un documento esistente. Un altro thread aspetta i nuovi messaggi.

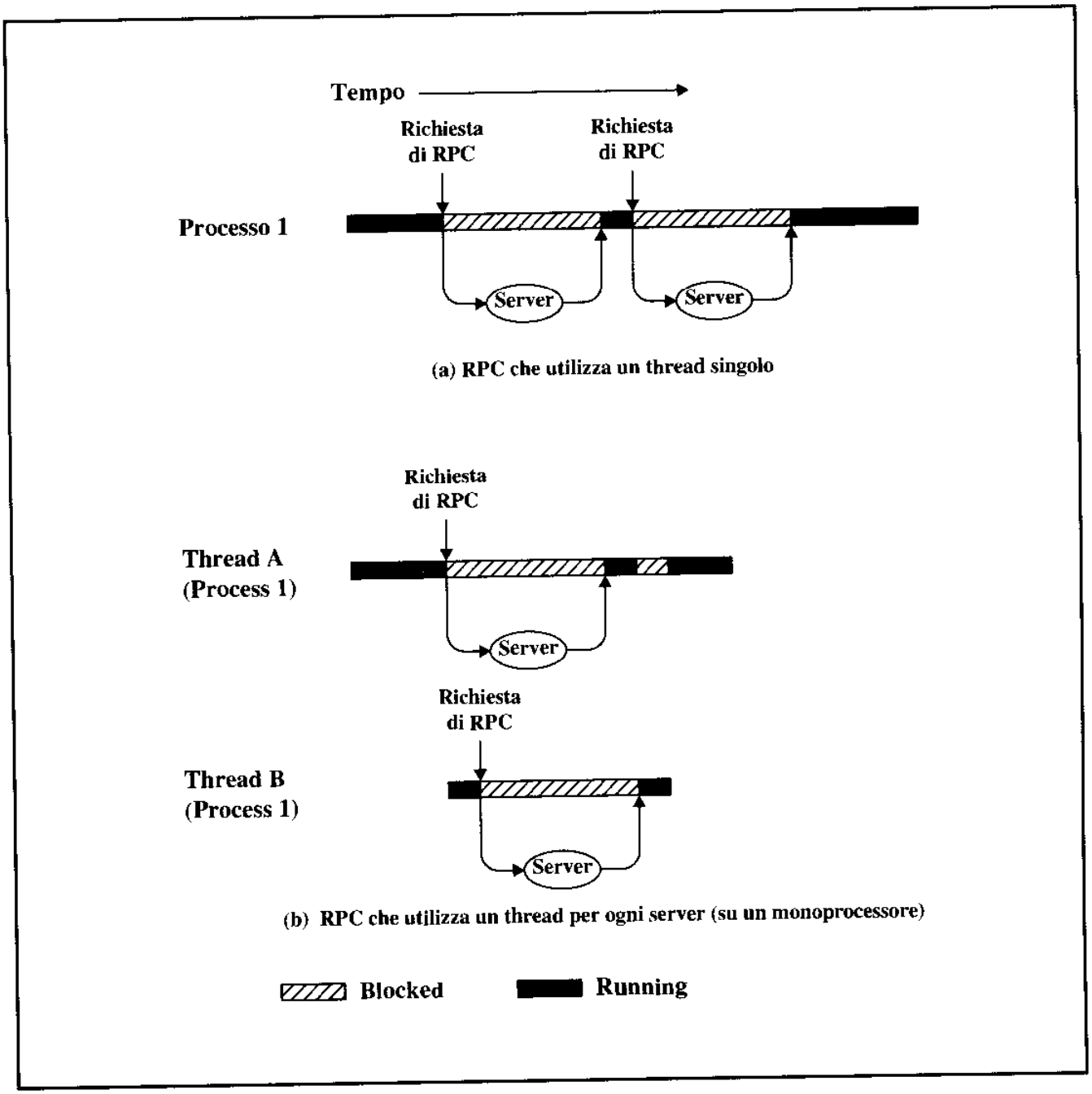


Figura 4.3 *Chiamata di procedura remota utilizzando i thread*

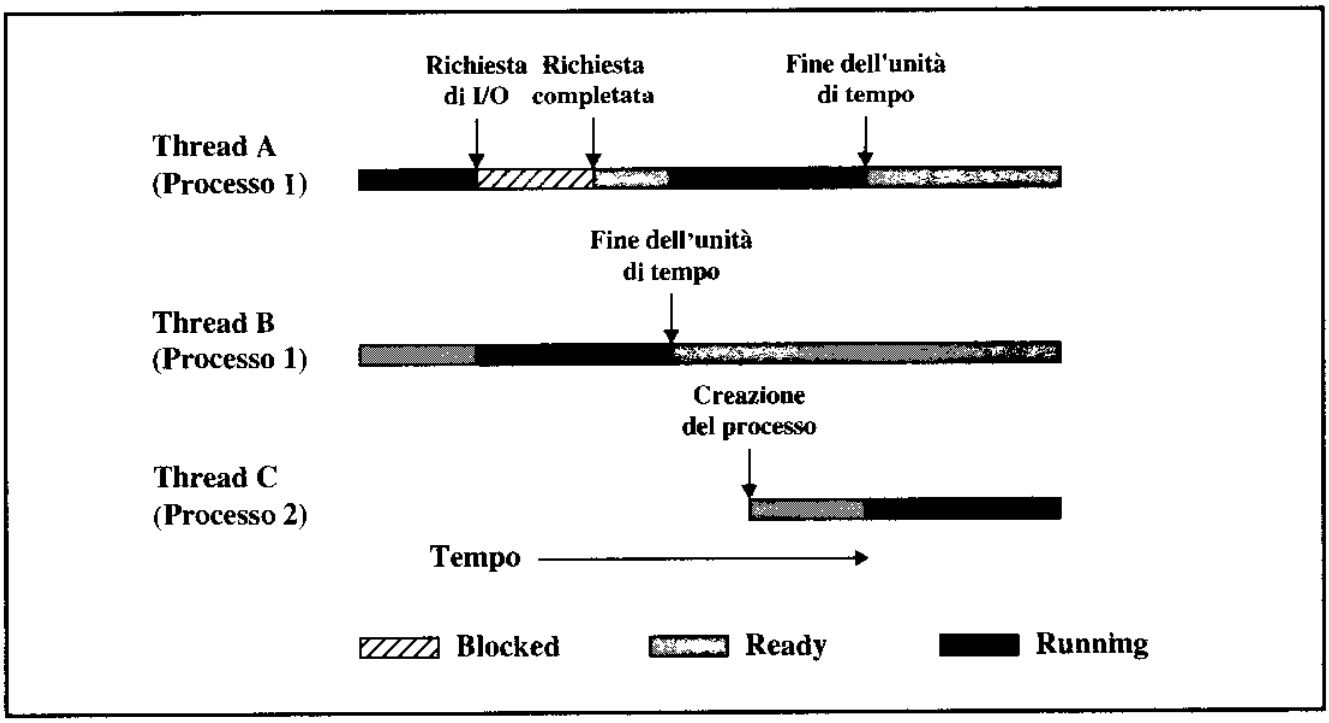


Figura 4.4 *Esempio di multithreading su un sistema monoprocessore*

Sincronizzare il thread di servizio e il thread che gestisce gli eventi è complicato, perché un utente può continuare a scrivere con la tastiera o muovere il mouse, attivando il thread di gestione degli eventi, mentre il thread di servizio è occupato. Quando avviene questo conflitto, PageMaker filtra questi messaggi e accetta solo alcuni messaggi di base, come il ridimensionamento della finestra.

Quando il thread di servizio completa un compito, invia un messaggio; fino a quel momento, l'attività dell'utente di PageMaker è limitata. Il programma indica questo stato disabilitando le voci di menu e mostrando un cursore "busy" (occupato). L'utente è libero di passare a un'altra

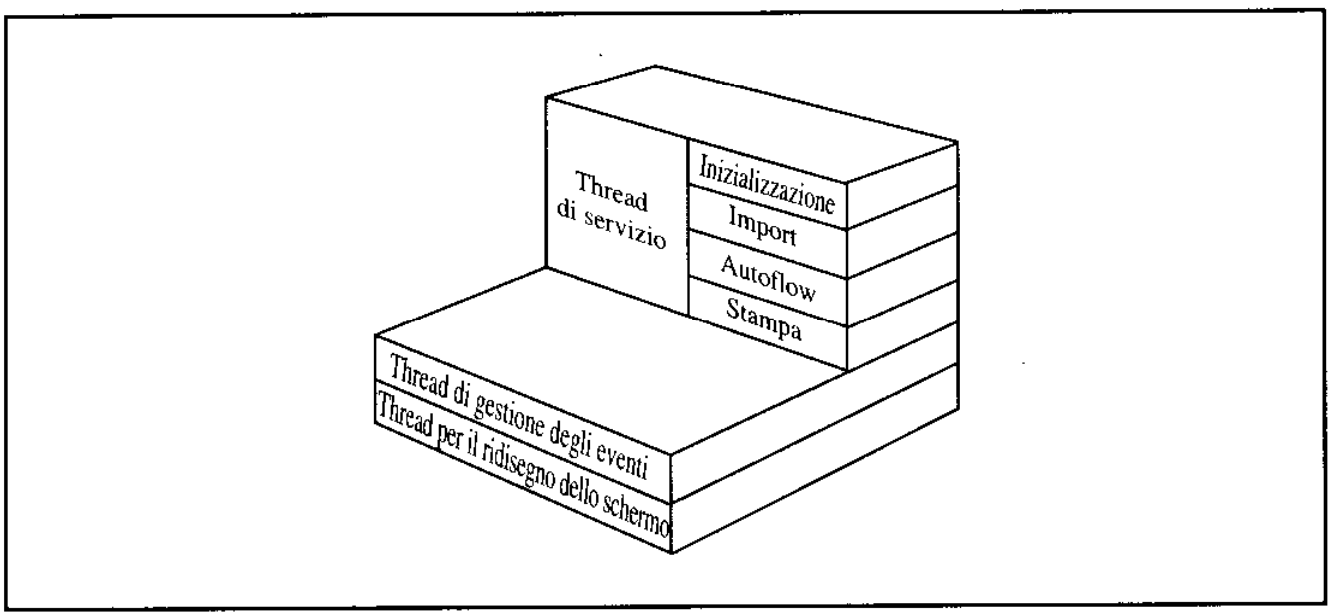


Figura 4.5 *Struttura dei thread di Aldus PageMaker*

applicazione, e quando il cursore "busy" si sposta sopra ad un'altra finestra, riprende l'aspetto normale per quell'applicazione.

Il ridisegno dello schermo viene effettuato da un thread a parte per due ragioni:

1. PageMaker non limita il numero di oggetti che possono apparire su una pagina; quindi, elaborare una richiesta di ridisegno può facilmente superare la soglia di 0.1 secondi.
2. L'uso di un thread separato permette all'utente di annullare l'operazione di ridisegno. In tal caso, quando un utente riscala una pagina, il ridisegno può iniziare immediatamente. Il tempo di risposta del programma è maggiore se ogni pagina deve essere disegnata completamente al fattore di scala vecchio, e poi ridisegnata con il nuovo fattore di scala.

Lo scrolling (*scorrimento*) dinamico – ridisegno dello schermo mentre l'utente sposta la barra di scorrimento – è anche possibile. Il thread che gestisce gli eventi, tiene sotto controllo la barra di scorrimento e ridisegna i righelli indicanti i margini (che possono essere disegnati velocemente per dare all'utente una sensazione immediata della posizione). Nel frattempo, il thread di ridisegno dello schermo cerca continuamente di ridisegnare la pagina.

Implementare il ridisegno dinamico senza l'uso di thread multipli, aggiungerebbe all'applicazione il peso del controllo della presenza di eventuali messaggi in vari punti. Il multithreading permette di separare le attività concorrenti del codice in maniera più naturale.

Thread a livello di utente e di kernel

Ci sono due categorie di implementazioni di thread: thread a livello utente (ULT, *user-level threads*) e thread a livello di kernel (KLT, *kernel-level threads*) [3]. In letteratura i secondi vengono anche detti thread supportati dal kernel, o processi leggeri.

Thread a livello di utente

In caso di ULT puro, tutto il lavoro di gestione dei thread viene effettuato dall'applicazione e il kernel non è conscio della presenza dei thread; la Figura 4.6a mostra questo approccio. Qualunque applicazione può essere programmata con thread multipli utilizzando una libreria per i thread, che è un insieme di routine per la gestione di ULT. La libreria contiene codice per creare e distruggere thread, per scambiare messaggi e dati fra thread, per schedulare l'esecuzione dei thread, e per salvare e ricaricare i contesti dei thread.

A meno di indicazioni contrarie, un'applicazione inizia con un singolo thread, associato ad un unico processo gestito dal kernel. In ogni momento dell'esecuzione dell'applicazione (quando il processo è in stato Running), l'applicazione può creare un nuovo thread relativo allo stesso processo. La creazione viene effettuata chiamando *spawn*, l'utilità di creazione della libreria per i thread, che riceve il controllo tramite una chiamata di procedura. La libreria per i thread crea una struttura dati per il nuovo thread, e passa il controllo ad uno degli altri thread dello stesso processo che si trova nello stato Ready, utilizzando qualche algoritmo di schedulazione. Quando

[3] Gli acronimi ULT e KLT compaiono solamente in questo libro, e sono introdotti per brevità.

viene passato il controllo alla libreria, il contesto del thread corrente viene salvato, e quando il controllo è passato dalla libreria ad un thread, il contesto di quel thread viene ristabilito. Il contesto si compone essenzialmente del contenuto dei registri utente, il program counter e gli stack pointer.

Tutta l'attività descritta nel paragrafo precedente avviene nello spazio utente e all'interno di un singolo processo. Il kernel non è conscio di questa attività, e continua a schedulare i processi come delle unità e assegna ad ogni processo un singolo stato di esecuzione (Ready, Running, Blocked, ecc.). L'esempio seguente dovrebbe chiarire la relazione fra schedulazione di thread e schedulazione di processi. Supponendo che il thread 3 del processo B sia in esecuzione, si possono verificare i seguenti casi:

- L'applicazione eseguita nel thread 3 effettua una chiamata di sistema che blocca B. Per esempio, una chiamata di I/O. Quindi il controllo passa al kernel, che chiama l'azione di I/O, mette il processo B in stato Blocked e attiva un altro processo. Nel frattempo, dal punto di vista della struttura dati mantenuta dalla libreria per i thread, il thread 3 del processo B è ancora in stato Running. È importante notare che il thread 3 in realtà non è in esecuzione su un processore fisico, ma è considerato in stato Running dalla libreria.
- L'interrupt generato dal clock passa il controllo al kernel, che stabilisce se il processo correntemente in esecuzione (B) ha esaurito la sua unità di tempo. Il kernel mette B in stato Ready e attiva un altro processo. Nel frattempo, dal punto di vista della struttura dati mantenuta dalla libreria per i thread, il thread 3 del processo B è ancora in stato Running.

In entrambi i casi precedenti, quando il kernel passa il controllo al processo B, l'esecuzione riparte dal thread 3. Inoltre, si noti che un processo può essere interrotto, perché ha esaurito la sua unità di tempo, o perché c'è un processo con priorità più alta, mentre sta eseguendo codice della libreria per i thread: in particolare, durante il cambio tra due thread. Quando poi il processo viene riattivato, l'esecuzione continua all'interno della libreria per i thread, che completa il cambio fra thread e trasferisce il controllo ad un nuovo thread dello stesso processo.

Ci sono vari vantaggi nell'uso di ULT rispetto a KLT:

1. Il cambio fra thread non richiede privilegi in modalità kernel, perché tutta la gestione delle strutture dati dei thread avviene nello spazio di indirizzamento utente di un singolo processo. Quindi, il processo non passa in modalità kernel per effettuare la gestione dei thread. Ciò risparmia il sovraccarico di due cambiamenti di modalità (da utente a kernel e da kernel a utente).
2. La schedulazione può essere diversa per ogni applicazione: un'applicazione può trarre il massimo vantaggio da un semplice algoritmo di schedulazione di tipo round-robin, un'altra da un algoritmo basato su priorità. L'algoritmo di schedulazione può essere ottimizzato in base all'applicazione, senza disturbare lo schedulatore del sistema operativo.
3. Gli ULT si possono eseguire su qualunque sistema operativo, non è necessario cambiare il kernel sottostante. La libreria per i thread è un insieme di utilità a livello di applicazione, condivise da tutte le applicazioni.

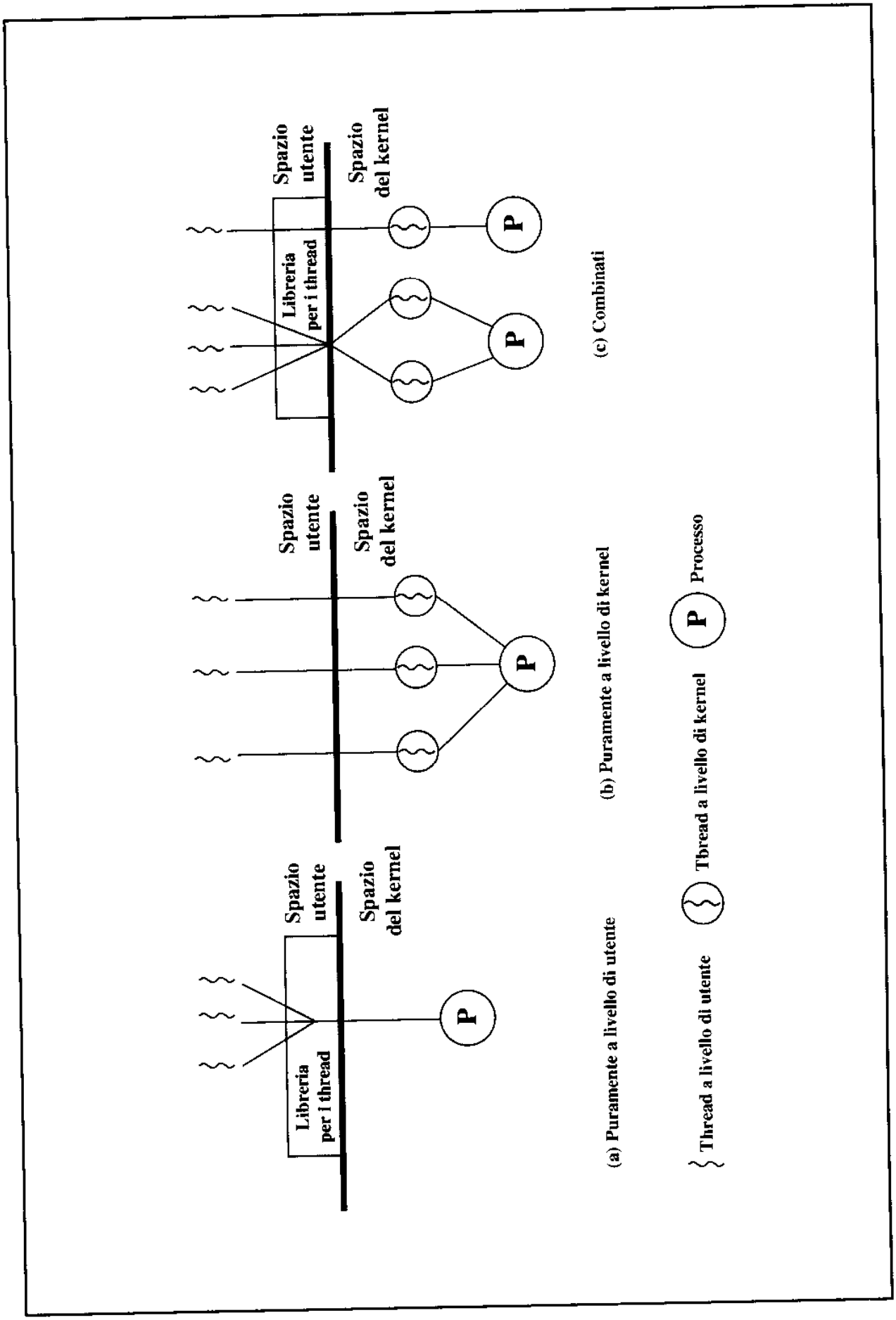


Figura 4.6 *Thread a livello di utente e a livello di kernel*

Ci sono due svantaggi degli ULT rispetto ai KLT:

1. In un tipico sistema operativo, la maggior parte delle chiamate di sistema comporta un blocco. Così, quando un thread esegue una chiamata di sistema, non viene bloccato solo quel thread, ma tutti quelli appartenenti allo stesso processo.
2. In una strategia puramente ULT, una applicazione multithread non può sfruttare il multiprocessing; un kernel assegna un processo ad un solo processore alla volta, quindi, in un dato istante, un solo thread per processo è in esecuzione. In realtà, si ha multiprogrammazione a livello di applicazione all'interno di un singolo processo, il che può produrre un aumento significativo delle prestazioni dell'applicazione; ci sono però applicazioni che sarebbero avvantaggiate se potessero eseguire porzioni di codice concorrentemente.

Ci sono modi di aggirare questi problemi: per esempio, si possono superare entrambi i problemi scrivendo un'applicazione composta da processi multipli, anziché thread multipli; tale approccio però elimina il vantaggio principale dei thread: ogni cambio di processo diventa un cambio tra processi anziché un cambio fra thread, causando un peggioramento delle prestazioni.

Un altro modo per superare il problema del blocco dei thread è di usare una tecnica nota come *jacketing*. Lo scopo del jacketing è di convertire una chiamata di sistema bloccante in una non bloccante. Ad esempio, invece di chiamare direttamente una routine di sistema per l'I/O, un thread chiama una routine "jacket" per l'I/O a livello utente. All'interno di questa routine si verifica se il dispositivo di I/O è occupato: in caso affermativo, il thread entra in stato Ready e passa il controllo (tramite la libreria per i thread) ad un altro thread; quando più tardi questo thread viene riattivato, controlla nuovamente il dispositivo di I/O.

Thread a livello di kernel

In caso di KLT puro, tutto il lavoro di gestione dei thread viene effettuato dal kernel. Nell'area dell'applicazione non c'è codice di gestione dei thread, ma semplicemente una API (Application Programming Interface, *Interfaccia per la Programmazione delle Applicazioni*) per la componente del kernel che gestisce i thread. Windows NT e OS/2 sono esempi di questo approccio.

La Figura 4.6b mostra l'approccio KLT puro. È possibile programmare qualunque applicazione come multithread; tutti i thread di un'applicazione sono supportati all'interno di uno stesso processo. Il kernel mantiene informazioni sia sul contesto del processo, sia sul contesto di ogni suo thread, ed effettua la schedulazione a livello di thread. Questo approccio supera i due punti deboli principali dell'approccio ULT. Primo, il kernel può schedulare simultaneamente vari thread dello stesso processo su vari processori. Secondo, se un thread di un processo è bloccato, il kernel può schedulare un altro thread dello stesso processo. Un altro vantaggio dell'approccio KLT è che le routine stesse del kernel possono essere multithread.

Lo svantaggio principale dell'approccio KLT rispetto all'approccio ULT è che il trasferimento del controllo da un thread ad un altro all'interno dello stesso processo richiede il passaggio in modalità kernel. Per illustrare le differenze, la Tabella 4.1 mostra i risultati di misurazioni effettuate con una macchina VAX monoprocessore su un sistema operativo tipo UNIX. I due

Tabella 4.1 *Tempi di latenza delle operazioni dei thread (μs) [ANDE92]*

Operazione	Thread a livello di utente	Thread a livello di kernel	Processi
Null Fork	34	948	11300
Signal-Wait	37	441	1840

benchmark sono Null Fork, che misura il tempo richiesto per creare, schedulare, eseguire e completare un processo/thread che chiama una procedura fittizia (cioè il sovraccarico dato dalla creazione di un processo/thread), e Signal-Wait, che misura il tempo impiegato da un processo/thread per mandare un segnale ad un processo/thread in attesa, ed aspettare che si verifichi una condizione (cioè il sovraccarico della sincronizzazione di due processi/thread). Per mettere nel giusto contesto questi numeri, si consideri che una chiamata di procedura sul VAX utilizzato in questi studi richiede circa 7 μs, mentre una trap del kernel richiede circa 17 μs. Si vede che c'è una differenza di un ordine di grandezza, o anche di più, fra ULT e KLT, e fra KLT e processi.

Quindi, mentre l'uso di multithreading KLT comporta un aumento di prestazioni significativo rispetto all'uso di processi con un solo thread, utilizzando ULT si ha un ulteriore aumento significativo. Comunque, la realizzazione o meno del secondo aumento di prestazioni dipende dalla natura delle applicazioni considerate. Se la maggior parte dei cambiamenti di contesto fra thread in una certa applicazione richiede un accesso in modalità kernel, è possibile che uno schema basato su ULT non abbia prestazioni molto migliori rispetto ad uno basato su KLT.

Approcci misti

Alcuni sistemi operativi sono una combinazione di ULT e KLT (Figura 4.6c); Solaris è l'esempio principale. In un sistema misto, la creazione dei thread è effettuata completamente nello spazio utente, e lo stesso accade per la schedulazione e la sincronizzazione in un'applicazione. Gli ULT multipli di una singola applicazione vengono mappati in un numero (minore o uguale) di KLT. Il programmatore può fissare il numero di KLT per una particolare applicazione e macchina, per raggiungere i risultati migliori.

In un approccio misto, vari thread della stessa applicazione possono essere eseguiti in parallelo su vari processori, e una chiamata di sistema bloccante non blocca necessariamente l'intero processo. Se progettato nel modo giusto, questo approccio dovrebbe combinare i vantaggi degli approcci ULT e KLT puri e minimizzare gli svantaggi.

Altri approcci

Come abbiamo detto, i concetti di unità di allocazione delle risorse e unità di allocazione sono tradizionalmente contenuti nel solo concetto di processo; in altre parole c'è una relazione 1:1 fra thread e processi. Recentemente c'è stato molto interesse nel fornire più thread all'interno dello stesso processo: una relazione molti-a-uno. Comunque, come mostra la Tabella 4.2, anche le altre due possibilità sono state analizzate: una relazione molti-a-molti e una relazione uno-a-molti.

Tabella 4.2 *Relazioni tra thread e processi*

Thread:Processi	Descrizione	Sistemi
1:1	Ogni thread di esecuzione è un processo unico con il proprio spazio di indirizzamento e le proprie risorse	Molte implementazioni di UNIX
M:1	Ogni processo ha associato un proprio spazio di indirizzamento e delle risorse. In ogni processo si possono creare ed eseguire molti thread.	Windows NT, Solaris, OS/2, OS/390, MACH
1:M	Un thread può spostarsi da un processo all'altro; ciò permette di spostare facilmente i thread fra sistemi diversi.	Ra(Clouds), Emerald
M:M	Combina le proprietà degli approcci M:1 e 1:M.	TRIX

Relazioni molti-a-molti

L'idea di avere una relazione molti-a-molti fra thread e processi è stata esplorata nel sistema operativo sperimentale TRIX [SIEB83, WARD80]. In TRIX ci sono i concetti di dominio e di thread; un dominio è un'entità statica, composta da uno spazio di indirizzamento e da "porte" attraverso cui si possono trasmettere e ricevere messaggi. Un thread è un singolo percorso di esecuzione, con uno stack di esecuzione, uno stato del processore e con informazioni di schedulazione.

Come nell'approccio multithreading appena discusso, thread multipli possono essere eseguiti in un singolo dominio, fornendo l'aumento di efficienza discusso prima. Comunque, è anche possibile eseguire in multitasking una singola attività o applicazione dell'utente. In tal caso, c'è un thread che può spostarsi da un dominio ad un altro.

L'uso di un singolo thread in domini multipli sembra motivato principalmente dal desiderio di fornire al programmatore degli strumenti di strutturazione. Per esempio, si consideri un programma che utilizza un sottoprogramma di I/O; in un ambiente di multiprogrammazione che permetta all'utente di creare processi, il programma principale potrebbe generare un nuovo processo per gestire l'I/O e continuare l'esecuzione. Comunque, se i passi successivi del programma dipendono dal risultato dell'operazione di I/O, allora il programma principale dovrà aspettare che l'altro programma di I/O termini. Ci sono molti modi di implementare questa applicazione:

1. L'intero programma può essere implementato come singolo processo. Questa è una soluzione semplice e ragionevole, che porta degli svantaggi per quanto riguarda la gestione della memoria. È possibile che l'intero processo richieda parecchia memoria principale, mentre il sottoprogramma di I/O richiederà uno spazio relativamente piccolo, per i propri buffer di I/O e per una quantità relativamente piccola di codice. Poiché il programma di I/O viene esegui-

to nello spazio di indirizzamento del programma più grande, o l'intero processo rimane in memoria durante l'operazione di I/O, oppure il codice del programma di I/O viene salvato su disco. Questa inefficienza nella gestione della memoria si verificherebbe anche se il programma principale e il sottoprogramma di I/O fossero implementati come due thread separati nello stesso spazio di indirizzamento.

2. Il programma principale e il sottoprogramma di I/O possono essere implementati come due processi separati. In questo caso c'è il sovraccarico della creazione del processo figlio. Se l'attività di I/O è frequente, bisogna o lasciare vivo il processo figlio, consumando così risorse, oppure creare e distruggere frequentemente il sottoprogramma, il che è inefficiente.

3. Trattare il programma principale e il sottoprogramma di I/O come un'attività singola da implementare come thread singolo. Comunque, è possibile creare uno spazio di indirizzamento (dominio) per il programma principale e uno per il sottoprogramma di I/O. Così, il thread può essere mosso fra i due spazi di indirizzamento durante l'esecuzione; il sistema operativo può gestire i due spazi di indirizzamento indipendentemente e non c'è il sovraccarico della creazione dei processi. Inoltre, lo spazio di indirizzamento usato dal sottoprogramma di I/O potrebbe essere condiviso con altri semplici programmi di I/O.

L'esperienza dei progettisti di TRIX indica che la terza opzione ha dei vantaggi e può essere la soluzione più efficiente per alcune applicazioni.

Relazioni uno-a-molti

Nel campo dei sistemi operativi distribuiti (progettati per controllare sistemi di computer distribuiti), si è interessati al concetto di thread principalmente come entità che può spostarsi fra diversi spazi di indirizzamento[4]. Un esempio notevole di questa ricerca è il sistema operativo Clouds, e specialmente il kernel, chiamato Ra [DASG92]. Un altro esempio è il sistema Emerald [STEE95].

Dal punto di vista dell'utente, un thread in Clouds è una unità di attività, mentre un processo è uno spazio di indirizzamento virtuale con associato un Process Control Block. Dopo la creazione, un thread inizia l'esecuzione in un processo chiamando un punto di entrata nel programma, in tale processo. I thread si possono muovere da uno spazio di indirizzamento ad un altro e in realtà si spostano da un computer all'altro. Quando un thread si sposta, deve portare con sé alcune informazioni, come il terminale di controllo, alcuni parametri globali e informazioni per lo schedulatore (ad esempio la priorità).

L'approccio di Clouds fornisce un modo efficace di isolare un utente e un programmatore dai dettagli dell'ambiente distribuito; l'attività di un utente si può rappresentare come un singolo thread, e il movimento di quel thread fra varie macchine può essere deciso dal sistema operativo per varie ragioni legate al sistema, come ad esempio la necessità di una risorsa remota, o un bilanciamento del carico.

---

[4] Lo spostamento di processi o thread fra diversi spazi di indirizzamento, o migrazione di thread, su diverse macchine è diventato importante negli ultimi anni; questo sarà argomento del Capitolo 14.

4.2 Multiprocessing simmetrico

Tradizionalmente il computer è visto come macchina sequenziale: la maggior parte dei linguaggi di programmazione richiede che il programmatore specifichi gli algoritmi come sequenze di istruzioni. Un processore esegue i programmi eseguendo istruzioni macchina in sequenza e una alla volta; ogni istruzione è eseguita tramite una sequenza di operazioni (prelevamento dell'istruzione, prelevamento degli operandi, esecuzione dell'operazione e memorizzazione dei risultati).

Questa visione del computer non è mai stata completamente corretta. A livello di microoperazioni, molti segnali di controllo vengono generati nello stesso tempo. La pipeline delle istruzioni, in cui almeno le operazioni di prelevamento ed esecuzione sono sovrapposte, è utilizzata da parecchio tempo. In entrambi gli esempi, alcune funzioni sono eseguite in parallelo.

Con l'evoluzione della tecnologia dei computer e con la diminuzione dei costi hardware, i progettisti hanno mostrato un maggior interesse per il parallelismo, generalmente per migliorare le prestazioni, e in alcuni casi per aumentare l'affidabilità. In questo libro vengono esaminati i due approcci più popolari per realizzare il parallelismo utilizzando più processori: multiprocessing simmetrico (SMP, *symmetric multiprocessing*) e cluster. SMP è discusso in questa sessione, i cluster sono esaminati nella Parte Sesta.

Architettura SMP

È utile vedere dove le architetture SMP si collocano nella categoria dei processori paralleli. La tassonomia introdotta da [FLYN72], che classifica i sistemi con processori paralleli, è ancora la più usata. Flynn ha proposto le seguenti categorie di sistemi di elaborazione:

- **Istruzione Singola, Dato Singolo** (SISD, *Single Instruction Single Data*): un singolo processore esegue una singola sequenza di istruzioni operando su dati memorizzati in una singola memoria.
- **Istruzione Singola, Dati Multipli** (SIMD, *Single Instruction Multiple Data*): una singola istruzione macchina controlla l'esecuzione simultanea di più elaborazioni, sincronizzate passo a passo. Ogni elemento di elaborazione ha associata una propria memoria per i dati, così ogni istruzione è eseguita su dati diversi da diversi processori. I processori vettoriali appartengono a questa categoria.
- **Istruzioni Multiple, Dato Singolo** (MISD, *Multiple Instruction Single Data*): una sequenza di dati viene trasmessa ad un insieme di processori, ognuno dei quali esegue una diversa sequenza di istruzioni. Questa architettura non è mai stata realizzata.
- **Istruzioni Multiple, Dati Multipli** (MIMD, *Multiple Instruction Multiple Data*): un insieme di processori esegue simultaneamente diverse sequenze di istruzioni su dati diversi.

Con l'architettura MIMD, i processori sono di tipo generico perché devono poter eseguire le istruzioni necessarie ad effettuare trasformazioni di dati generiche. I MIMD si possono ulterior-

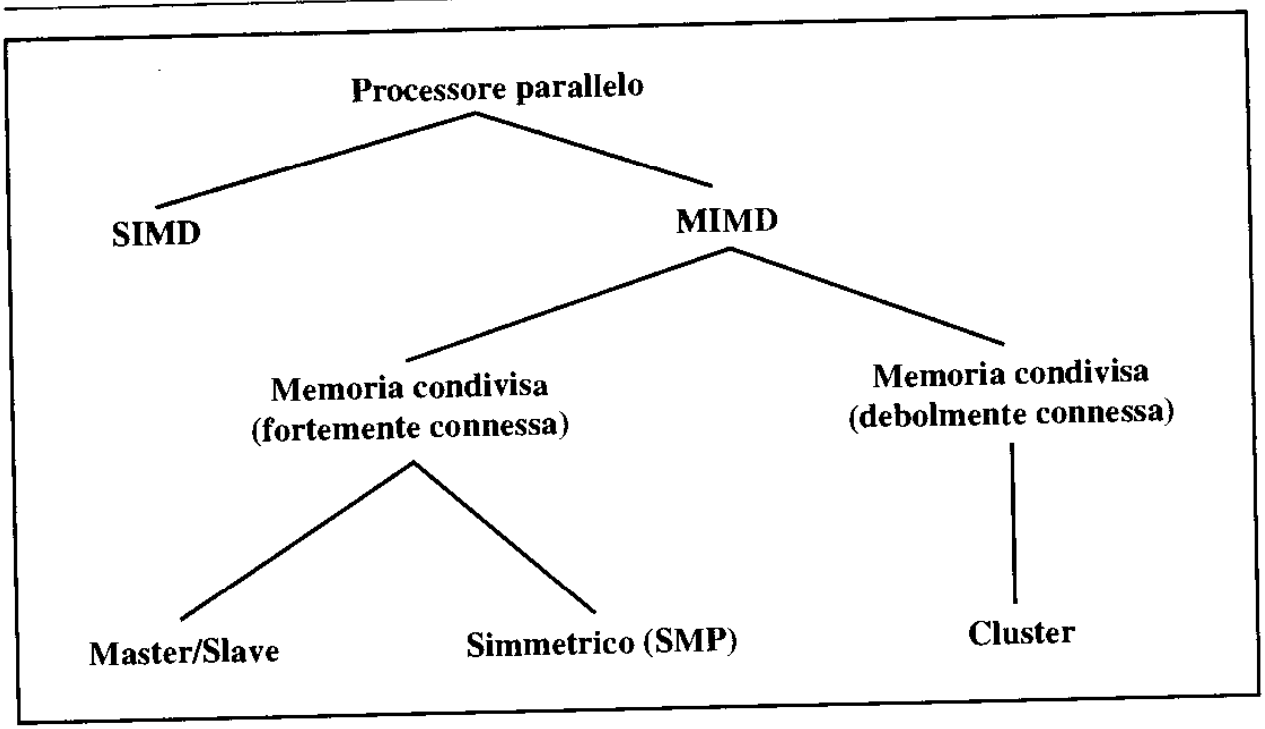


Figura 4.7 *Architetture di processori paralleli*

mente suddividere a seconda della modalità di comunicazione fra i processori (Figura 4.7). Se ogni processore ha una memoria dedicata, allora ogni unità di elaborazione è un computer indipendente. La comunicazione fra i computer può avvenire tramite percorsi fissi o tramite una rete; tali sistemi vengono detti **cluster**, o multicomputer. Se i processori condividono la memoria, allora ogni processore accede ai programmi e ai dati contenuti nella memoria condivisa, e la comunicazione avviene tramite la memoria; tale sistema viene detto **multiprocessore a memoria condivisa**.

Una classificazione generale di multiprocessori a memoria condivisa è basata su come i processi vengono assegnati ai processori. I due approcci fondamentali sono master/slave (*padrone/schiavo*) e simmetrico. Con un'architettura **master/slave** il kernel del sistema operativo risiede sempre su un processore particolare. Gli altri processori possono eseguire unicamente programmi utente e utilità del sistema operativo. Il master è responsabile della schedulazione dei processi o thread. Quando un processo/thread è attivo, se lo slave ha bisogno di un servizio (ad esempio una chiamata di I/O), deve mandare una richiesta al master e aspettare che il servizio venga effettuato. Questo approccio è piuttosto semplice e richiede modifiche limitate ad un sistema operativo monoprocessore con multiprogrammazione. La risoluzione dei conflitti è semplificata, poiché un solo processore ha il controllo di tutte le risorse di memoria e di I/O. Gli svantaggi sono i seguenti: (1) un fallimento del master interrompe l'intero sistema, (2) il master può diventare un collo di bottiglia per le prestazioni, perché è il solo che può effettuare la schedulazione e la gestione dei processi. In un **multiprocessore simmetrico**, il kernel può essere eseguito su ogni processore, e tipicamente ogni processore autogestisce la schedulazione dell'insieme di processi o thread disponibili. Il kernel può essere costituito da molti processi o molti thread, quindi le sue parti possono essere eseguite in parallelo. L'approccio SMP complica il sistema operativo: si deve garantire che due processori non scelgano lo stesso processo e che

i processi in coda non vengano persi. Ci sono tecniche che devono essere impiegate per risolvere e sincronizzare le richieste di risorse.

La progettazione di SMP e di cluster è complessa, e riguarda problemi di organizzazione fisica, strutture di interconnessione, comunicazione fra processori, progettazione del sistema operativo e tecniche per il software applicativo. In questo testo verranno analizzati i problemi di progettazione del sistema operativo, con cenni all'organizzazione, per i casi SMP e cluster.

Organizzazione SMP

La Figura 4.8 mostra l'organizzazione generale di un multiprocessore simmetrico. Ci sono vari processori, ognuno contenente la propria unità di controllo, unità aritmetico-logica e registri. Ogni processore ha accesso alla memoria condivisa e ai dispositivi di I/O tramite un meccanismo di interconnessione: di solito un bus. I processori comunicano tra loro attraverso la memoria (messaggi e informazione di stato vengono messi in indirizzi di memoria condivisi); i processi possono anche scambiarsi segnali direttamente. Solitamente la memoria è organizzata in modo tale che sia possibile effettuare simultaneamente più accessi allo stesso blocco.

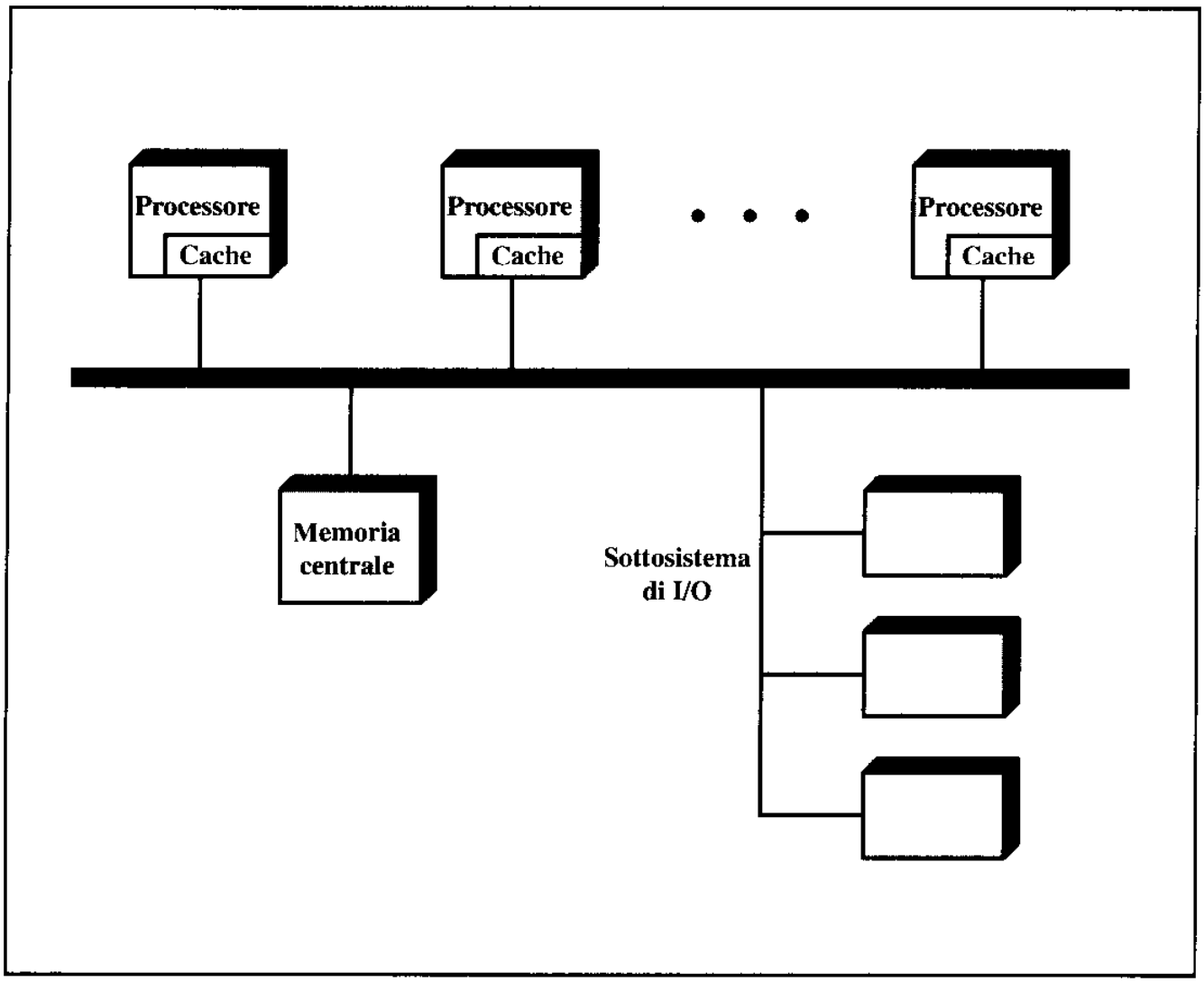


Figura 4.8 *Organizzazione di un multiprocessore simmetrico*

Nelle macchine moderne, generalmente i processori hanno almeno un livello di memoria cache privata. Tale uso della cache introduce ulteriori problemi di progettazione. Poiché ogni cache locale contiene l'immagine di una porzione di memoria principale, se una parola viene modificata in una cache, potrebbe invalidare una parola in un'altra cache. Per impedire ciò, gli altri processori devono essere informati che è stata effettuata una modifica: tale problema viene detto *problema di coerenza della cache* e tipicamente è risolto in hardware anziché dal sistema operativo[5].

Considerazioni per la progettazione di sistemi operativi multiprocessore

Un sistema operativo SMP gestisce i processori e le altre risorse in maniera tale che la visione dell'utente sia di un sistema monoprocessore con multiprogrammazione; infatti, un utente può realizzare applicazioni che fanno uso di processi o thread multipli senza preoccuparsi del numero di processori disponibili. Così un sistema operativo multiprocessore deve fornire tutte le funzionalità di un sistema di multiprogrammazione, più la gestione di processori multipli. Le caratteristiche fondamentali per il progettista sono le seguenti:

- **Processi o thread concorrenti:** le routine del kernel devono essere rientranti per permettere ai vari processori di eseguire simultaneamente lo stesso codice del kernel. Avendo processori multipli che eseguono le stesse parti o parti diverse del kernel, le tabelle del kernel e le strutture di gestione devono essere manipolate accuratamente in modo da evitare stalli o operazioni non valide.
- **Schedulazione:** può essere effettuata da qualunque processore, così è necessario evitare conflitti. Se viene usato multithreading a livello di kernel, è possibile schedulare più thread dello stesso processo simultaneamente su più processori. La schedulazione multiprocessore è esaminata nel Capitolo 10.
- **Sincronizzazione:** poiché vari processi attivi possono accedere allo spazio di indirizzamento condiviso o alle risorse di I/O comuni, diventa fondamentale realizzare la sincronizzazione, grazie alla quale si ottengono la mutua esclusione e l'ordinamento degli eventi. Un meccanismo di sincronizzazione tipicamente usato nei sistemi operativi multiprocessore è quello dei lock, descritti nel Capitolo 5.
- **Gestione della memoria:** un multiprocessore presenta tutti i problemi di gestione della memoria visti nella Parte Terza per le macchine monoprocessore; inoltre, il sistema operativo deve sfruttare il parallelismo hardware, ad esempio le memorie multiporta, per raggiungere prestazioni ottimali. I meccanismi di paginazione di diversi processori devono essere coordinati per garantire la coerenza quando diversi processori condividono una pagina o un segmento, e stabilire quali pagine rimpiazzare.

---

[5] [STAL96] contiene una descrizione degli schemi per la coerenza della cache in hardware.

- **Affidabilità e tolleranza dei guasti:** il sistema operativo deve permettere un decadimento graduale in caso di fallimento dei processori. Lo schedulatore e altre porzioni del sistema operativo devono riconoscere la perdita di un processore e ristrutturare le tabelle di gestione in maniera appropriata.

Poiché generalmente i metodi di progettazione di un sistema operativo multiprocessore sono estensioni delle soluzioni adottate per la multiprogrammazione su monoprocessori, non verranno fatte trattazioni separate, ma i metodi specifici per i multiprocessori saranno trattati nei contesti opportuni nel resto del libro.

4.3 Microkernel

Negli ultimi tempi il concetto di microkernel ha ricevuto molta attenzione: un microkernel è un piccolo nucleo di sistema operativo che fornisce le basi per estensioni modulari. Il termine è spesso abusato, e ci sono aspetti dei microkernel che sono trattati diversamente da parte di progettisti diversi. Tipicamente, ci si chiede quanto il kernel debba essere piccolo per qualificarsi come microkernel, come progettare i driver dei dispositivi per ottenere le migliori prestazioni mantenendo un'astrazione dallo hardware, se le operazioni esterne al microkernel debbano essere eseguite nello spazio kernel o in spazio utente, e se mantenere il codice esistente per i sottosistemi (ad esempio una versione di UNIX), o riscriverlo da zero.

L'uso del microkernel è divenuto popolare con il sistema operativo Mach; in teoria, questo approccio fornisce un alto livello di flessibilità e modularità. Un altro uso ben pubblicizzato del microkernel è Windows NT, i cui benefici dichiarati contengono la portabilità oltre alla modularità. Il microkernel è circondato da un insieme di sottosistemi compatti, in modo da facilitare l'implementazione di NT su varie piattaforme. Attualmente molti altri prodotti vantano un'implementazione con microkernel, e questo approccio verrà probabilmente utilizzato in futuro nella maggior parte dei sistemi operativi per personal computer, workstation e server.

Architettura del microkernel

I primi sistemi operativi sviluppati dalla metà alla fine degli anni '50 erano solitamente progettati con scarsa attenzione per la strutturazione: nessuno allora aveva esperienza nella costruzione di sistemi software di grandi dimensioni, e i problemi causati da dipendenze reciproche e dalle interazioni di componenti erano sottostimati grossolanamente. In questi **sistemi operativi monolitici**, ciascuna procedura in pratica poteva chiamare ogni altra procedura. Tale mancanza di strutturazione divenne insostenibile con la crescita delle dimensioni dei sistemi. Ad esempio, la prima versione di OS/360 fu creata da 5000 programmatori in un periodo di 5 anni e conteneva oltre un milione di linee di codice; Multics, sviluppato in seguito, raggiungeva 20 milioni di linee di codice. Come discusso nella Sezione 2.3, le tecniche di programmazione modulare erano necessarie per gestire lo sviluppo di software di questa scala. Specificamente, si svilupparono **sistemi operativi a strati** (Figura 4.9a) in cui le funzioni sono organizzate in maniera gerarchica e c'è interazione solamente fra stati adiacenti. Quindi è difficile implemen-

tare versioni ad hoc di un sistema operativo di base per mezzo dell'aggiunta o rimozione di alcune funzioni, ed è difficile garantire la sicurezza a causa delle molte interazioni fra strati adiacenti.

Ricordiamo che anche questa terminologia non è applicata coerentemente in letteratura. Il termine *sistema operativo monolitico* viene spesso usato facendo riferimento ai due tipi di sistemi operativi che qui vengono chiamati *monolitici* e *a strati*.

La filosofia del **microkernel** è che solo le funzioni assolutamente essenziali del nucleo del sistema operativo dovrebbero essere nel kernel; i servizi meno essenziali e le applicazioni sono costruiti sopra al microkernel e vengono eseguiti in modalità utente. Sebbene la linea di separazione fra cosa è dentro e cosa è fuori dal microkernel vari da un progetto all'altro, la caratteristica comune è che molti servizi che tradizionalmente facevano parte del sistema operativo divengono sottosistemi esterni, che interagiscono con il kernel e tra di loro; essi comprendono i driver dei dispositivi, i file system, il gestore della memoria virtuale, il sistema a finestre e i servizi di sicurezza.

L'architettura a microkernel sostituisce la tradizionale stratificazione verticale dei sistemi operativi con una orizzontale (Figura 4.9b). I componenti del sistema operativo esterni al microkernel sono implementati come processi server; essi interagiscono fra di loro su base di parità, tipicamente tramite passaggio di messaggi attraverso il microkernel. Così, il microkernel gestisce lo scambio di messaggi: valida i messaggi, li passa fra i componenti e concede i permessi di accesso allo hardware.

Per esempio, se un'applicazione vuole aprire un file, manda un messaggio al server del file system; se vuole creare un processo o un thread, manda un messaggio al server dei processi. I vari server possono scambiarsi messaggi e chiamare le funzioni primitive del microkernel: ciò costituisce un'architettura client/server all'interno di un singolo computer.

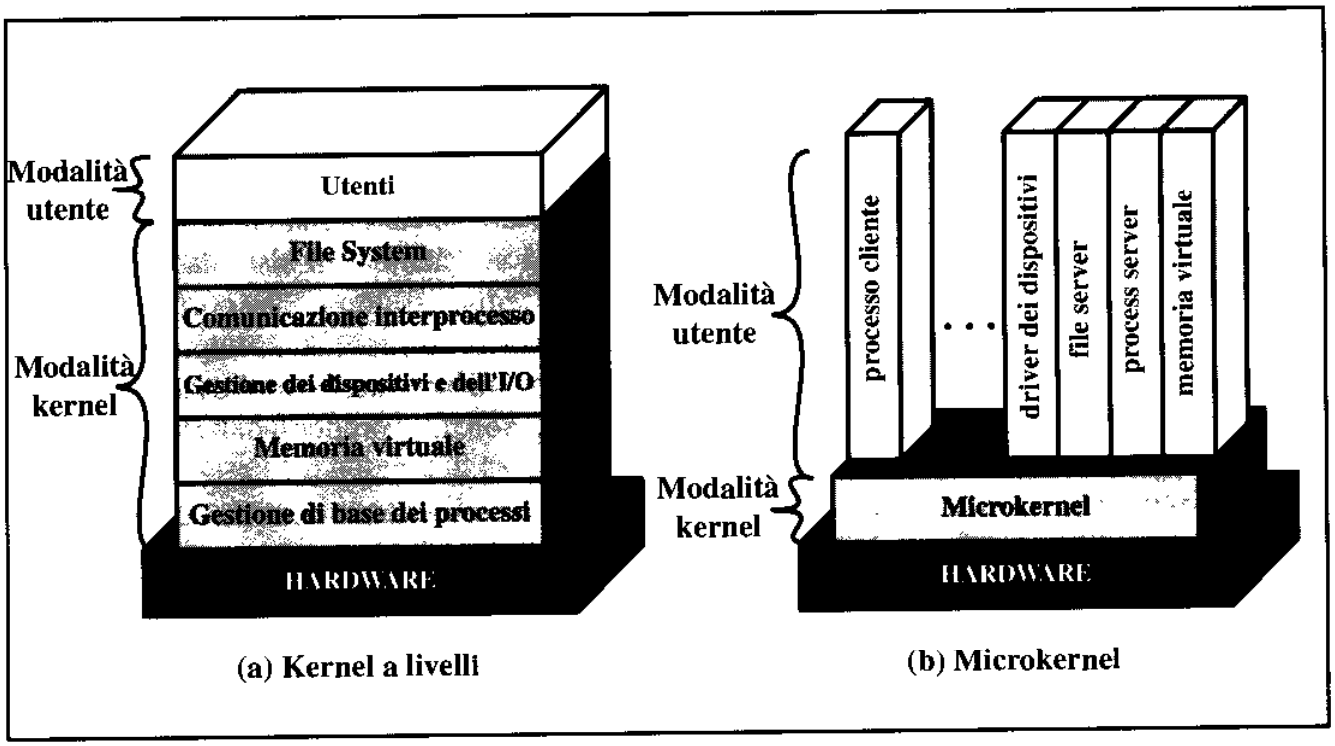


Figura 4.9 *Architettura del kernel*

Benefici di un'organizzazione a microkernel

In letteratura ([FINK97], [LIED96a], [WAYN94a]) si trovano descritti i vantaggi derivanti dall'uso di microkernel:

- Interfaccia uniforme
- Estensibilità
- Flessibilità
- Portabilità
- Affidabilità
- Supporto ai sistemi distribuiti
- Sistemi operativi orientati agli oggetti (OOOS, *Object-Oriented Operating Systems).*

La progettazione dei microkernel impone una **interfaccia uniforme** alle richieste da parte dei processi. I processi non devono fare distinzione fra servizi a livello di kernel e a livello utente, perché ogni servizio è fornito tramite passaggio di messaggi.

Inevitabilmente ogni sistema operativo dovrà essere esteso con nuove caratteristiche, quando verranno sviluppati nuovi dispositivi hardware e tecniche software. L'architettura microkernel facilita l'**estensibilità**, permettendo l'aggiunta di nuovi servizi, nonché la disponibilità di molti servizi all'interno della stessa area funzionale. Ad esempio, ci potrebbero essere diverse organizzazioni dei file per i dischetti; ogni organizzazione potrebbe essere implementata come processo a livello utente, invece di avere molti servizi per i file all'interno del kernel. Quindi, gli utenti possono scegliere, fra i vari servizi, quelli che meglio soddisfano le esigenze del caso. Con l'architettura microkernel, quando si aggiunge una nuova caratteristica, è necessario modificare o aggiungere solamente i server che la riguardano; l'impatto di un nuovo server o di una modifica è ristretto a un sottoinsieme del sistema. Inoltre, le modifiche non richiedono la creazione di un nuovo kernel.

Una caratteristica legata all'estensibilità dell'architettura microkernel è la **flessibilità**. Non solo è possibile aggiungere caratteristiche al sistema, ma alcune caratteristiche possono essere rimosse per ottenere un'implementazione più snella ed efficiente. Un sistema operativo a microkernel non è necessariamente un sistema piccolo; infatti la strutturazione porta ad aggiungere un grande numero di caratteristiche, ma non tutti hanno bisogno, ad esempio, di un alto livello di sicurezza o la possibilità di effettuare elaborazioni distribuite. Se molte caratteristiche (dal punto di vista della richiesta di memoria) sono rese opzionali, il prodotto di base interesserà un maggior numero di utenti.

Il quasi-monopolio di Intel in molti segmenti del mercato dei computer non è destinato a durare per sempre, così la **portabilità,** in un sistema operativo, diventa una caratteristica interessante. Nell'architettura microkernel tutto il codice che dipende dal tipo di processore è contenuto nel kernel, così le modifiche necessarie a trasferire il sistema su un nuovo processore sono minori, e si possono suddividere in gruppi logici.

Maggiore è la dimensione di un prodotto software, più difficile è garantire l'**affidabilità**. Sebbene una progettazione modulare aiuti a migliorare l'affidabilità, si possono ottenere mi-

glioramenti ancora più significativi con un'architettura microkernel; inoltre, un microkernel piccolo può essere testato rigorosamente. L'uso di un piccolo numero di API aumenta la probabilità di produrre codice di qualità per i servizi del sistema operativo esterni al kernel. Il programmatore di sistema deve imparare un numero limitato di API e ha possibilità limitate di interagire con altre componenti del sistema e comprometterne il funzionamento.

Il microkernel si presta al **supporto ai sistemi distribuiti**, come i cluster controllati da un sistema operativo distribuito. Quando un messaggio viene inviato da un processo cliente ad un server, il messaggio deve contenere un identificatore del servizio richiesto. Se un sistema distribuito (ad esempio un cluster) è configurato in maniera tale che tutti i processi e i servizi abbiano identificatore unico, allora in effetti c'è un'unica immagine del sistema a livello di microkernel; un processo può inviare un messaggio senza conoscere la macchina su cui il destinatario risiede. Si ritornerà su questo punto nella discussione sui sistemi distribuiti nella Parte Sesta.

Un'architettura microkernel funziona bene nel contesto dei **sistemi operativi orientati agli oggetti**: tale approccio può disciplinare la progettazione del microkernel e lo sviluppo di estensioni modulari del sistema operativo; di conseguenza, alcuni progetti di microkernel si stanno muovendo nella direzione dei sistemi orientati agli oggetti [WAYN94b]. Un approccio promettente alla fusione dell'architettura microkernel con i principi dei sistemi operativi orientati agli oggetti è l'uso di componenti [MESS96]: essi sono oggetti con interfacce ben definite, e si possono interconnettere per formare software in stile scatola di montaggio. Tutte le interazioni fra componenti utilizzano le interfacce dei componenti. Altri sistemi, come Windows NT, non si affidano esclusivamente a metodi orientati agli oggetti, ma incorporano principi object-oriented nel progetto del microkernel.

Prestazioni del microkernel

Uno svantaggio potenziale dei microkernel, spesso citato, è quello delle prestazioni. Costruire un messaggio, inviarlo, accettarlo e decodificare la risposta, richiede più tempo che effettuare una singola chiamata di servizio. Comunque, altri fattori entrano in gioco, pertanto è difficile in generale valutare un eventuale decadimento di prestazioni.

Molto dipende dalla dimensione e dalle funzionalità del microkernel. [LIED96a] riassume alcuni studi che rivelano uno scadimento di prestazioni sostanziale per i cosiddetti microkernel di prima generazione. Questi problemi permangono nonostante gli sforzi per ottimizzarne il codice. Una risposta al problema è di estendere il microkernel reintegrando nel sistema operativo i server critici e i driver. Gli esempi principali di questo approccio sono Mach e Chorus. Aggiungendo alcune funzionalità chiave al microkernel si riduce il numero di cambiamenti di stato utente-kernel e il numero di cambiamenti di spazio di indirizzamento. Comunque, il decadimento di prestazioni viene ridotto a spese dei punti di forza del progetto di un microkernel: interfacce minime, flessibilità, e così via.

Un altro approccio è di non espandere il microkernel, ma di ridurlo. [LIED96b] sostiene che, se progettato adeguatamente, un microkernel molto piccolo elimina il decadimento di prestazioni e aumenta la flessibilità e l'affidabilità. Per dare un'idea delle dimensioni, un tipico microkernel di prima generazione si componeva di 300 KB di codice e 140 chiamate di sistema. Un esempio di piccolo microkernel di seconda generazione è L4 [LIED95], che si compone di 12 KB di

codice e 7 chiamate di sistema. L'esperienza su tali sistemi indica che si possono ottenere prestazioni pari o migliori di quelle di un sistema operativo a strati come UNIX.

Progettazione del microkernel

Poiché diversi microkernel hanno funzionalità e dimensioni diverse, non esistono regole facili per decidere quali funzioni vadano fornite dal microkernel e quali strutture vadano implementate. In questa sezione si presenta un insieme minimo di funzioni e di servizi, per dare un'idea della progettazione del microkernel.

Il microkernel deve contenere le funzioni che dipendono direttamente dallo hardware e quelle necessarie per supportare i server e le applicazioni in modalità utente. Queste funzioni cadono nelle categorie generali di: gestione primitiva della memoria, comunicazione tra processi (IPC, *InterProcess Communication*) e gestione degli interrupt e dell'I/O.

Gestione primitiva della memoria

Il microkernel controlla lo spazio di indirizzamento per rendere possibile l'implementazione della protezione a livello di processo. Finché il microkernel è responsabile di mappare ogni pagina virtuale in una pagina fisica, il grosso della gestione della memoria si può implementare all'esterno del kernel, compresa la protezione dello spazio di indirizzamento di un processo dagli altri processi, l'algoritmo di sostituzione delle pagine e altre logiche di paginazione. Ad esempio, un modulo per la gestione della memoria virtuale esterno al microkernel decide quando portare una pagina in memoria e quale pagina già in memoria va rimpiazzata; il microkernel mappa i riferimenti a queste pagine in un indirizzo fisico della memoria centrale.

L'idea di implementare la paginazione e la gestione della memoria virtuale al di fuori del kernel fu introdotta dal paginatore esterno di Mach [YOUN87]: la Figura 4.10 ne illustra l'operato. Quando un thread di un'applicazione accede ad una pagina che non si trova in memoria centrale, viene generato un page fault (*errore di pagina*) e l'esecuzione salta al kernel. Il kernel

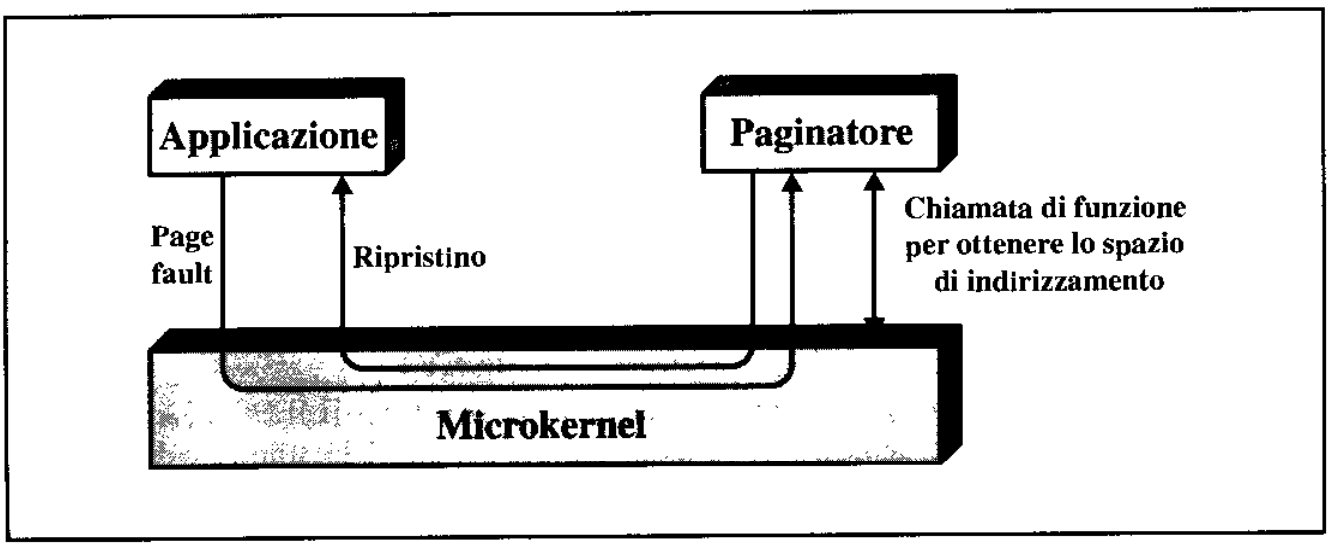


Figura 4.10 *Gestione dei page fault*

invia un messaggio al processo che effettua la paginazione indicando quale pagina è stata richiesta. Quest'ultimo può decidere di caricare la pagina e allocare spazio (un *frame*) per contenerla; il kernel ed il processo gestore delle pagine interagiscono per mappare le operazioni logiche del paginatore nella memoria fisica, ed infine, quando la pagina è disponibile, il paginatore manda un messaggio all'applicazione.

Questa tecnica permette ad un processo esterno al kernel di mappare file e basi di dati nello spazio di indirizzamento dell'utente senza chiamare il kernel; inoltre, eventuali politiche di condivisione della memoria, specifiche di un'applicazione, possono essere implementate fuori dal kernel.

[LIED95] suggerisce un insieme di sole tre operazioni del microkernel per supportare la paginazione esterna e la gestione della memoria virtuale:

- **Grant:** il proprietario di uno spazio di indirizzamento (un processo) può concedere alcune sue pagine ad un altro processo. Il kernel rimuove queste pagine dallo spazio di indirizzamento del proprietario e le assegna al processo ricevente.
- **Map:** un processo può mappare qualunque sua pagina nello spazio di indirizzamento di un altro processo, così entrambi i processi hanno accesso alle pagine; ciò crea condivisione di memoria fra i due processi. Il kernel mantiene l'assegnazione delle pagine al proprietario originale, ma permette all'altro processo di accedervi.
- **Flush:** un processo può chiedere indietro qualunque pagina abbia concesso o mappato ad un altro processo.

All'avvio del sistema, il kernel definisce l'intera memoria fisica come un singolo spazio di indirizzamento, controllato da un processo di sistema. Quando vengono creati nuovi processi, le pagine dello spazio di indirizzamento originale possono essere concesse o mappate ai nuovi processi; tale schema gestisce simultaneamente diversi schemi di memoria virtuale.

Comunicazione interprocesso

La forma basilare di comunicazione fra processi o thread in un sistema operativo a microkernel è il messaggio; esso contiene un'intestazione che identifica il processo mittente e il ricevente, e un corpo che contiene i dati, un puntatore ad un blocco di dati, o informazioni di controllo sul processo. Tipicamente, si può pensare alla comunicazione interprocesso come basata su porte associate ai processi: una porta è, essenzialmente, una coda di messaggi destinati ad un particolare processo, a cui si associa una lista di accessibilità (*capability*), che indica quali processi possono comunicare con questo processo. L'identità della porta e l'accessibilità sono amministrate dal kernel; un processo può concedere l'accesso alla propria porta mandando un messaggio al kernel in cui indica la nuova accessibilità della porta.

I/O e gestione degli interrupt

Con un'architettura microkernel, è possibile gestire interrupt hardware come messaggi e contenere le porte di I/O nello spazio di indirizzamento. Il microkernel può riconoscere gli

interrupt, ma non può gestirli; invece, genera un messaggio per il processo a livello utente associato a quell'interrupt al momento. Così, quando un interrupt viene abilitato, un particolare processo a livello utente è assegnato all'interrupt e il kernel mantiene la mappatura. La trasformazione degli interrupt in messaggi deve essere effettuata dal microkernel, ma questo non è coinvolto nella gestione degli interrupt specifica dei dispositivi.

[LIED96a] suggerisce di vedere lo hardware come un insieme di thread che hanno identificatori di thread unici e mandano messaggi (contenenti semplicemente l'identità del thread) a thread associati nello spazio utente. Il thread ricevente stabilisce se il messaggio proviene da un interrupt, e da quale interrupt in particolare. La struttura generale di tale codice a livello utente è la seguente:

```
driver thread:
  do
       wait(msg, mittente);
       if mittente = mio_interrupt_hardware
            then leggi/scrivi le porte di I/O;
                 azzera l'interrupt hardware
            else ...
       endif
enddo
```

4.4 Thread e SMP in Windows NT

La progettazione di processi in Windows NT si basa sulla necessità di fornire il supporto ad ambienti di sistema diversi. I processi supportati da diversi sistemi operativi differiscono sotto molti aspetti, tra cui:

- Come sono chiamati i processi.
- Presenza o meno di thread all'interno dei processi.
- Come sono rappresentati i processi.
- Come vengono protette le risorse dei processi.
- Quali meccanismi vengono usati per la comunicazione interprocesso e per la sincronizzazione.
- Come sono correlati i processi.

Quindi, le strutture per i processi e i servizi forniti dal kernel di NT sono relativamente semplici e generici, permettendo ad ogni sottosistema del sistema operativo di emulare una particolare struttura o funzionalità per i processi. Le caratteristiche importanti dei processi di NT sono le seguenti:

- I processi di NT sono implementati come oggetti.
- Un processo eseguibile può contenere uno o più thread.
- Gli oggetti di tipo thread e di tipo processo incorporano capacità di sincronizzazione.

La Figura 4.11 illustra il modo in cui un processo è correlato alle risorse che controlla o che usa. Ad ogni processo viene assegnato un token di accesso per la sicurezza, chiamato token primario; quando un utente si connette per la prima volta, NT crea un token di accesso che contiene un identificatore di sicurezza per l'utente. Ogni processo, creato dall'utente o eseguito per suo conto, possiede una copia del suo token di accesso. NT usa il token per verificare se l'utente ha il permesso di accedere agli oggetti "sicuri" o ad eseguire particolari funzioni sul sistema o sugli oggetti sicuri. Il token di accesso determina se il processo può cambiare i propri attributi; il processo non ha un handle (*maniglia*) aperto per il suo token di accesso, e se il processo tenta di aprire tale handle, il sistema di sicurezza stabilisce se ciò è permesso e quindi se il processo può cambiare i propri attributi.

Al processo è associata anche una serie di blocchi che definiscono lo spazio di indirizzamento virtuale assegnato ad esso al momento. Il processo non può modificare direttamente queste strutture, ma deve affidarsi al gestore della memoria virtuale, che gli fornisce un servizio di allocazione della memoria.

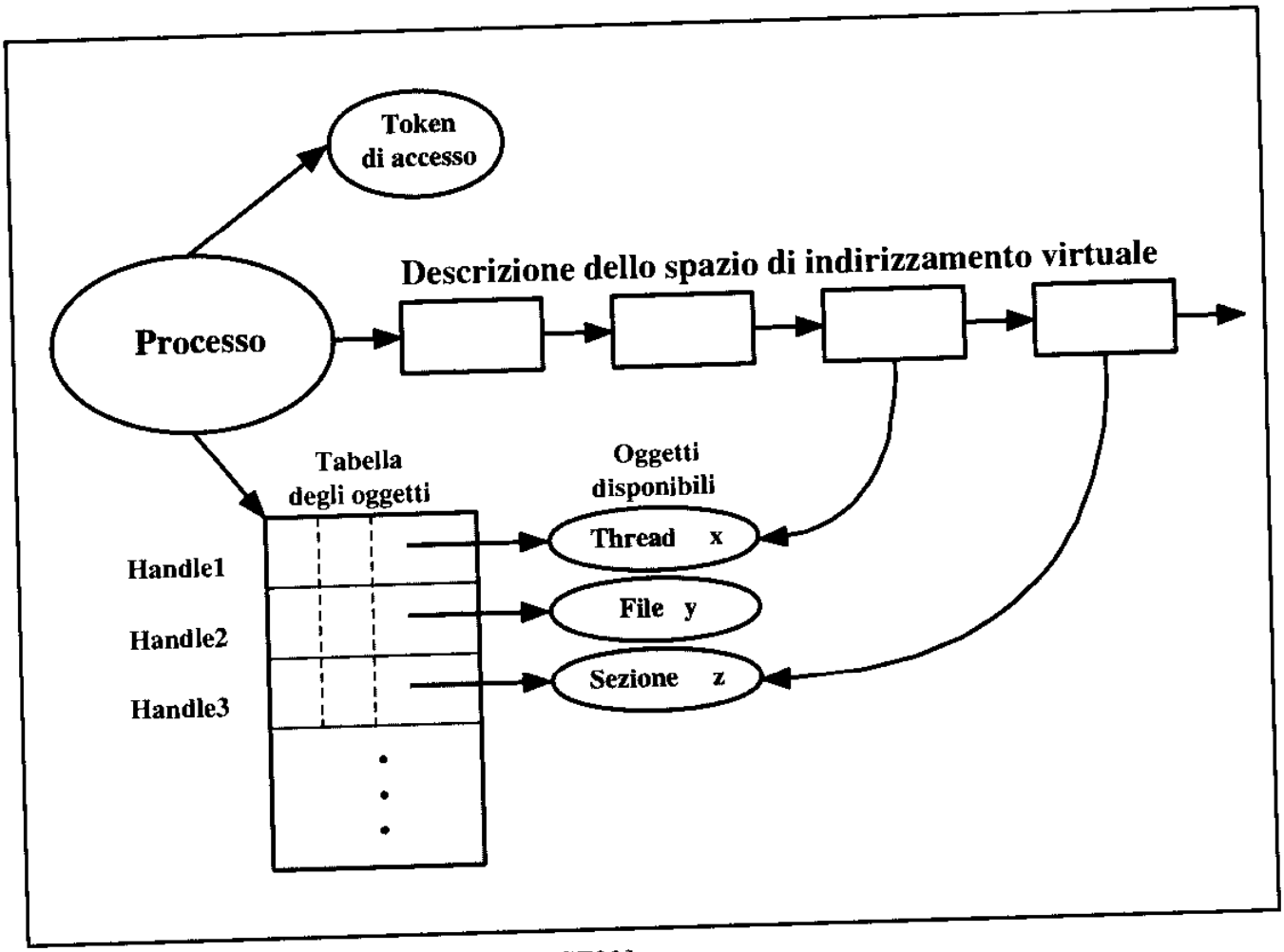


Figura 4.11 *Processi e risorse in NT [CUST93]*

Infine, il processo ha una tabella di oggetti, con handle ad altri oggetti che esso conosce; in particolare, vi si trova un handle per ogni thread in esso contenuto (nella Figura 4.11 si vede un unico thread); inoltre, il processo ha accesso ad un oggetto di tipo file e ad un oggetto che definisce una porzione di memoria condivisa.

Oggetti di tipo processo e di tipo thread

La struttura orientata agli oggetti di NT facilita lo sviluppo di una gestione dei processi di uso generico, usando due tipi di oggetti correlati ai processi: processi e thread. Un processo è un'entità corrispondente ad un job di un utente o ad un'applicazione, che possiede risorse, come la memoria, o può aprire file. Un thread è una unità allocabile, viene eseguito sequenzialmente, e può essere interrotto per permettere al processore di passare ad un altro thread.

Ogni processo di NT è rappresentato da un oggetto la cui struttura generale è descritta in Figura 4.12a. Ogni processo è definito da una lista di attributi, e contiene una lista di azioni, o

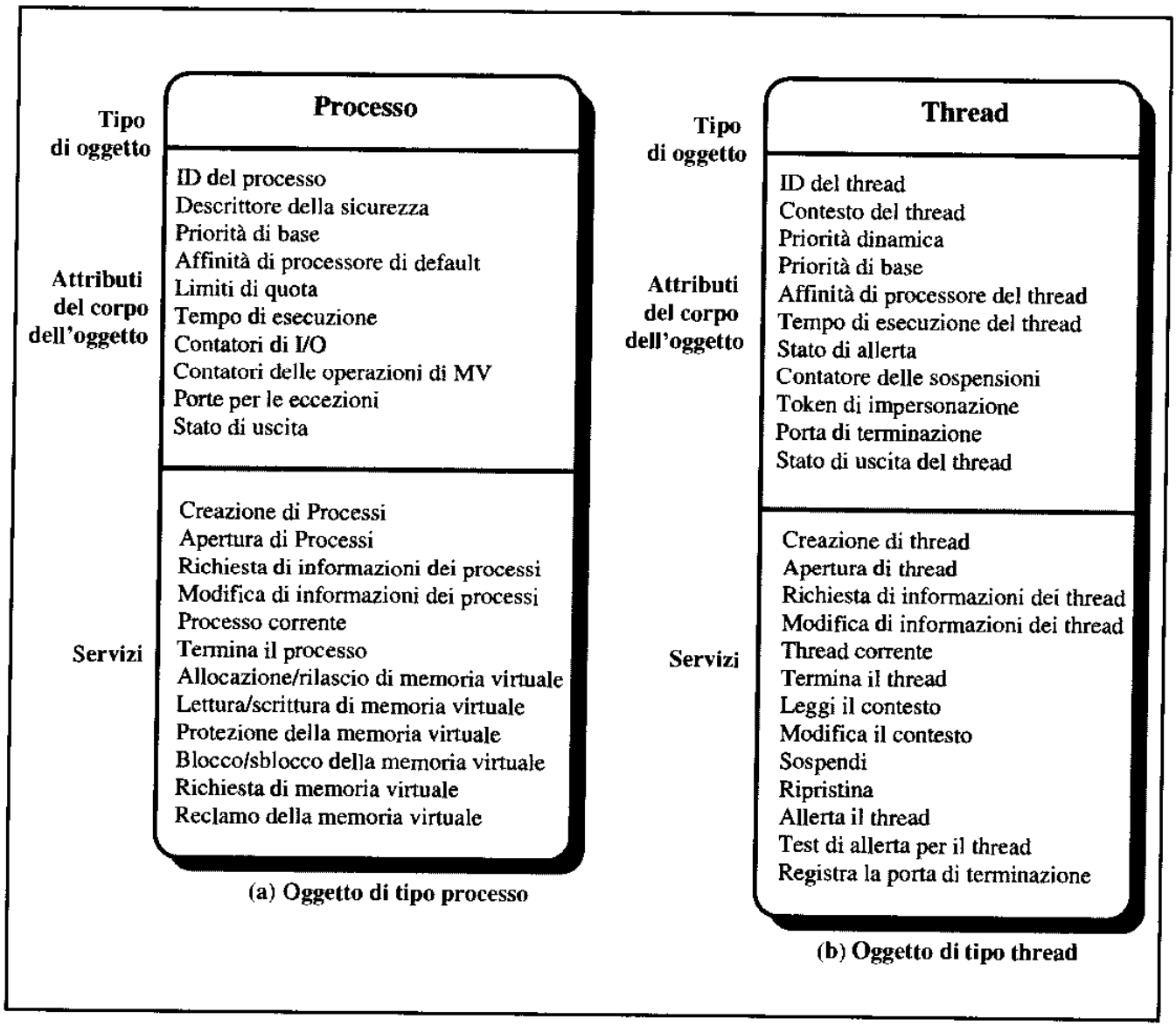


Figura 4.12 *Oggetti di tipo processo e di tipo thread in Windows NT*

servizi, che può effettuare. Un processo effettua un servizio quando riceve un messaggio appropriato; l'unico modo di chiamare tale servizio è tramite messaggi ad un oggetto di tipo processo che fornisce quel servizio. Quando NT crea un nuovo processo, usa la classe (o tipo) dell'oggetto, definita in NT come il modello per generare una nuova istanza dell'oggetto; al momento della creazione, sono assegnati i valori agli attributi. La Tabella 4.3 dà una breve definizione di ogni attributo degli oggetti di tipo processo.

Un processo di NT deve contenere almeno un thread per poter essere eseguito; tale thread può poi creare altri thread. In un sistema multiprocessore, thread multipli dello stesso processo possono essere eseguiti in parallelo; la Figura 4.12b mostra la struttura di un oggetto di tipo thread, e la Tabella 4.4 definisce gli attributi. Si noti che alcuni attributi dei thread somigliano a quelli dei processi: in tali casi, il valore dell'attributo del thread è derivato da quello del processo. Ad esempio, l'affinità processori/thread è l'insieme dei processori, in un sistema multiprocessore, che possono eseguire un thread, ed è uguale o un sottoinsieme dell'affinità processi/processori.

Si noti che uno degli attributi di un oggetto di tipo thread è il contesto. Questa informazione permette la sospensione e riattivazione dei thread; inoltre è possibile modificare il comportamento di un thread alterandone il contesto quando è sospeso.

Tabella 4.3 *Attributi degli oggetti di tipo processo in Windows NT*

ID del processo	Valore unico che identifica il processo per il sistema operativo.
Descrittore della sicurezza	Descrive chi ha creato un oggetto, e chi ha e chi non ha accesso ad esso.
Priorità di base	Priorità di esecuzione di base dei thread del processo.
Affinità di processore di default	L'insieme di default di processori su cui i thread del processo possono essere eseguiti.
Limiti di quota	Quantità massima di memoria di sistema paginata e non paginata, occupazione del file delle risorse, e tempo di processore che il processo può usare.
Tempo di esecuzione	Tempo di esecuzione totale di tutti i thread del processo.
Contatori di I/O	Variabili che memorizzano il numero e il tipo delle operazioni di I/O effettuate dai thread del processo.
Contatori delle operazioni di MV	Variabili che memorizzano il numero e il tipo delle operazioni sulla memoria virtuale effettuate dai thread del processo.
Porte per le eccezioni	Canali di comunicazione interprocesso a cui il gestore dei processi manda un messaggio quando uno dei thread del processo provoca una eccezione.
Stato di uscita	La ragione della terminazione del processo.

Tabella 4.4 *Attributi degli oggetti di tipo thread in Windows NT*

ID del thread	Valore unico che identifica il thread quando effettua una chiamata al server.
Contesto del thread	Insieme dei valori dei registri e altri dati volatili che definiscono lo stato di esecuzione del thread.
Priorità dinamica	Priorità di esecuzione del thread in ogni istante.
Priorità di base	Limite inferiore della priorità di esecuzione in ogni istante.
Affinità di processore del thread	Insieme di processori sui quali il thread può essere eseguito; è un sottoinsieme dell'affinità di processore del processo a cui appartiene.
Tempo di esecuzione del thread	Tempo totale di esecuzione di un thread in modalità utente e in modalità kernel.
Stato di allerta	Flag che indica se il thread deve esegnire una chiamata di procedura asincrona.
Contatore delle sospensioni	Numero di volte che l'esecuzione del thread è stata sospesa senza essere ripristinata.
Token di impersonazione	Token temporaneo che permette ad un thread di effettuare delle operazioni al posto di un altro processo (usato dai sottosistemi).
Porta di terminazione	Canale di comunicazione interprocesso a cui il gestore dei processi manda un messaggio quando il thread termina (usato dai sottosistemi).
Stato di uscita del thread	La ragione della terminazione del thread.

Multithreading

NT supporta la concorrenza fra processi perché thread di processi diversi possono essere eseguiti concorrentemente. Inoltre, thread multipli dello stesso processo possono essere assegnati a processori separati ed essere eseguiti concorrentemente; in questo modo, si ha la concorrenza senza il sovraccarico dato dall'uso di processi multipli. I thread di un processo possono scambiarsi informazioni attraverso lo spazio di indirizzamento comune e avere accesso alle risorse condivise del processo; thread in processi diversi possono scambiarsi informazioni utilizzando la memoria condivisa, se è stata condivisa a livello di processo.

Un processo multithread orientato agli oggetti è un modo efficiente di implementare un'applicazione server; ad esempio, un processo server può servire molti clienti; ogni richiesta dei clienti provoca la creazione di un nuovo thread del server.

Stati dei thread

Un thread di NT, se esiste, è in uno dei sei stati seguenti (Figura 4.13):

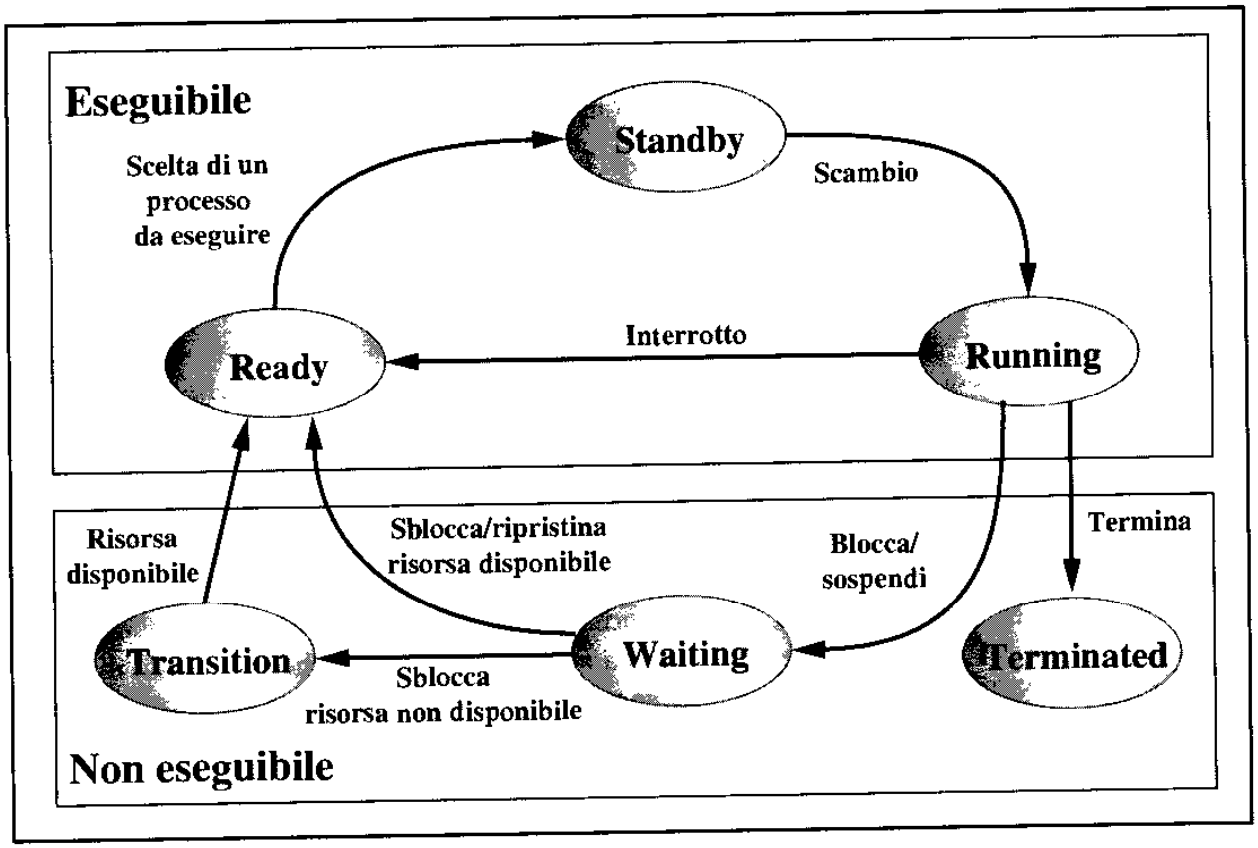


Figura 4.13 *Stati dei thread di NT [PHAM96]*

- **Ready:** può essere schedulato per l'esecuzione. L'allocatore del microkernel tiene traccia di tutti i thread in stato Ready e li schedula in ordine di priorità.
- **Standby:** un thread in stato Standby è stato scelto per un particolare processore; il thread aspetta in questo stato finché quel processore è disponibile. Se la priorità del thread in stato Standby è sufficientemente alta, il thread in esecuzione su quel processore può essere sostituito da quello in attesa; altrimenti, il thread in stato Standby aspetta finché il thread in esecuzione si blocca o termina il suo quanto di tempo.
- **Running:** quando il microkernel esegue un cambio di thread o di processo, il thread in standby entra in stato Running e comincia l'esecuzione, continuando finché non viene interrotto, oppure finisce il quanto di tempo, si blocca, o termina. Nei primi due casi ritorna nello stato Ready.
- **Waiting:** un thread entra in stato Waiting quando: (1) è bloccato su un evento (ad es. I/O), (2) aspetta volontariamente per ragioni di sincronizzazione, o (3) un sottosistema dell'ambiente forza il thread ad auto-sospendersi. Quando la condizione di attesa è soddisfatta, il thread passa allo stato Ready se tutte le sue risorse sono disponibili.
- **Transition:** un thread entra in questo stato dopo lo stato Waiting se è pronto per l'esecuzione, ma le risorse non sono disponibili. Ad esempio, lo stack del thread potrebbe essere stato inviato fuori dalla memoria centrale dal processo gestore della paginazione; quando le risorse diventano disponibili, il thread passa allo stato Ready.

- **Terminated:** un thread può terminare per sua richiesta, per richiesta di un altro thread, o quando il suo processo genitore termina. Quando le strutture del thread sono state eliminate, esso può essere rimosso dal sistema, oppure conservato per essere inizializzato nuovamente in futuro.

Supporto per i sottosistemi del sistema operativo

I meccanismi generali per la gestione dei processi e dei thread devono supportare le strutture dei thread particolari di vari sistemi operativi clienti. È responsabilità di ogni sottosistema sfruttare le caratteristiche dei thread e dei processi di NT per emulare le caratteristiche dei thread e dei processi dei sistemi operativi corrispondenti. Questa area di gestione dei processi/thread è piuttosto complicata, e ne verrà data solamente una breve panoramica.

La creazione dei processi inizia con la richiesta di un nuovo processo da parte di un'applicazione di un certo sistema operativo. La richiesta di creazione è inviata da una applicazione al sottosistema protetto corrispondente. Il sottosistema a sua volta manda una richiesta di processo a NT Executive, che crea un oggetto di tipo processo e ritorna al sottosistema un handle per l'oggetto. Quando NT crea un processo, non crea automaticamente anche un thread, ma, nel caso di Win32 e OS/2, un nuovo processo deve sempre essere creato con un thread: pertanto, per questi sistemi operativi, il sottosistema chiama nuovamente il gestore dei processi di NT per creare il thread per il nuovo processo, e ne riceve l'handle da NT. L'informazione appropriata relativa al thread ed al processo viene poi restituita all'applicazione. Nel caso di Windows a 16-bit e POSIX, i thread non sono supportati; quindi, per questi sistemi operativi, il sottosistema riceve da NT un thread per il nuovo processo, che può essere attivato, ma ritorna all'applicazione solamente l'informazione riguardante il processo, ed il fatto che il processo sia implementato utilizzando un thread non è visibile all'applicazione.

Quando un nuovo processo viene creato in Win32 o OS/2, il nuovo processo eredita molti degli attributi del processo creante. Comunque, nell'ambiente di NT, la creazione del processo viene effettuata indirettamente. Un processo applicativo cliente manda la richiesta di creazione di un processo al sottosistema corrispondente; poi un processo nel sottosistema manda una richiesta di processo ad NT Executive. Poiché l'effetto desiderato è che il nuovo processo erediti le caratteristiche del processo cliente e non quelle del processo server, NT permette che il sottosistema specifichi il genitore del nuovo processo; in questo modo esso eredita dal genitore il token di accesso, i limiti di quota, la priorità di base e l'affinità per difetto ai processori.

Supporto al multiprocessing simmetrico

NT supporta una configurazione hardware a multiprocessore simmetrico; i thread di ogni processo, compresi quelli di Executive, possono essere eseguiti su qualunque processore. In assenza di restrizioni di affinità, spiegate nel prossimo paragrafo, il microkernel assegna un thread in stato Ready al primo processore disponibile. Ciò assicura che nessun processore sia inattivo o stia eseguendo un thread a priorità bassa quando un thread ad alta priorità è Ready. Thread multipli dello stesso processo possono essere eseguiti simultaneamente su processori multipli.

A meno di indicazioni contrarie, il microkernel usa la politica di *affinità debole* nell'assegnare i thread ai processori: l'allocatore cerca di assegnare un thread in stato Ready allo stesso processore su cui è stato eseguito l'ultima volta: ciò consente il riutilizzo di dati ancora presenti, dall'esecuzione precedente, nella memoria cache del processore. Un'applicazione può restringere l'esecuzione dei suoi thread a certi processori (*affinità forte*).

4.5 Thread e gestione di SMP in Solaris

Solaris implementa un'insolita architettura multilivello dei thread, progettata per fornire particolare flessibilità per lo sfruttamento delle risorse del processore.

Architettura multithread

Solaris 2.x fa uso di quattro concetti distinti correlati ai thread:

- **Processi:** sono i normali processi UNIX e contengono lo spazio di indirizzamento dell'utente, lo stack e il Process Control Block.
- **Thread a livello utente:** sono implementati con una libreria per i thread nello spazio di indirizzamento di un processo, e sono invisibili al sistema operativo. I thread a livello di utente (ULT)[6] sono l'interfaccia per il parallelismo delle applicazioni.
- **Processi leggeri:** un processo leggero (LWP, lightweight process) si può vedere come una mappatura di ULT in thread del kernel: ogni processo leggero supporta uno o più ULT, ed è mappato in un thread del kernel. I processi leggeri sono schedulati dal kernel indipendentemente, e sui multiprocessori possono essere eseguiti in parallelo.
- **Thread del kernel:** sono entità fondamentali che possono essere schedulate e allocate su un processore del sistema.

La Figura 4.14 illustra le relazioni fra queste quattro entità: si noti che c'è sempre esattamente un thread del kernel per ogni processo leggero. Un processo leggero all'interno di un processo è visibile all'applicazione, in altri termini, le strutture dei processi leggeri sono nello spazio di indirizzamento del processo rispettivo. Inoltre, ogni processo leggero è associato ad un unico thread del kernel, e le strutture dati per quel thread sono tenute nello spazio di indirizzamento del kernel.

Nell'esempio, il processo 1 si compone di un singolo ULT legato ad un singolo processo leggero; in questo modo, c'è un unico thread di esecuzione corrispondente ad un processo UNIX tradizionale; le applicazioni usano questa struttura di processo quando non è richiesta concorrenza all'interno di un singolo processo. Il processo 2 corrisponde ad una strategia ULT pura, tutti gli ULT sono supportati dallo stesso thread del kernel, quindi è possibile eseguire un solo

[6] Ribadiamo che l'acronimo ULT è esclusivo di questo libro e non si trova nella letteratura di Solaris.

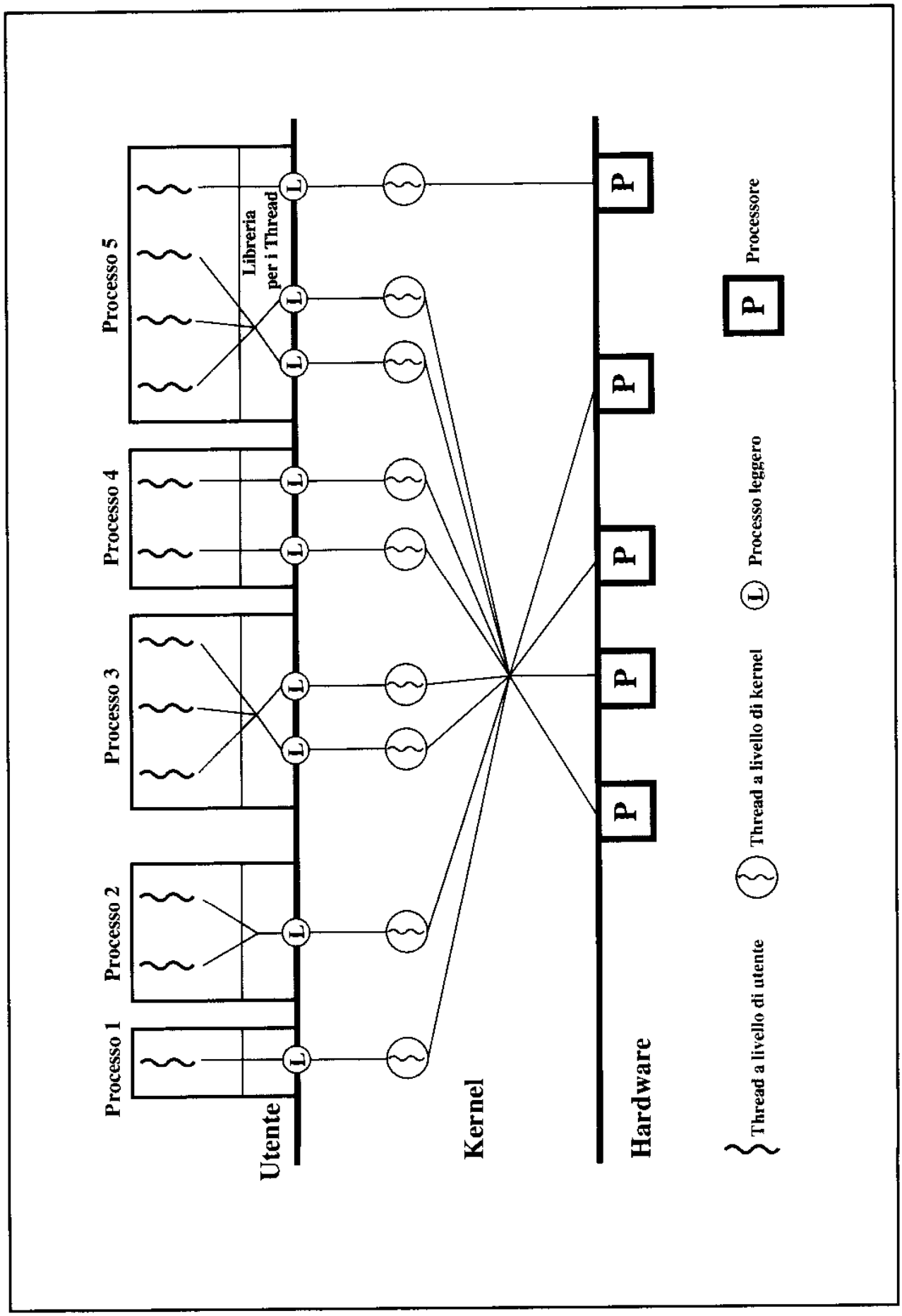


Figura 4.14 *Esempio di architettura multithread in Solaris*

ULT alla volta: in definitiva, questa struttura è utile per le applicazioni che richiedono concorrenza, ma non hanno bisogno di thread multipli in esecuzione. Il processo 3 mostra vari thread, distribuiti su un numero inferiore di processi leggeri: in generale, Solaris permette alle applicazioni di distribuire ULT su un numero minore o uguale di processi leggeri, e quindi di specificare, a livello del kernel, il grado di parallelismo che l'applicazione richiede al kernel. I thread del processo 4 sono permanentemente legati ai processi leggeri in una corrispondenza uno a uno: in questo modo si rende il parallelismo a livello del kernel pienamente visibile all'applicazione, il che è utile se tipicamente i thread verranno sospesi frequentemente per essere bloccati. Il processo 5 mostra sia una mappatura di ULT multipli in processi leggeri multipli, sia un ULT legato ad un processo leggero; inoltre, vi si trova un processo leggero legato ad un processore particolare.

La Figura 4.14 non mostra thread del kernel che non sono associati a processi leggeri, ma il kernel crea, esegue, e distrugge i propri thread per eseguire delle funzioni di sistema specifiche. L'uso di thread del kernel piuttosto che processi del kernel per implementare le funzioni del sistema riduce il sovraccarico di scambi di contesto all'interno del kernel (da un cambio di processo fra processi a un cambio interno al kernel).

Motivazioni

La combinazione di thread a livello utente e a livello di kernel dà al programmatore delle applicazioni l'opportunità di esplorare la concorrenza nel modo più efficiente e adatto per una data applicazione.

Alcuni programmi possiedono dei parallelismi logici che si possono sfruttare per semplificare e strutturare il codice, ma non necessitano di parallelismo in hardware. Ad esempio, una applicazione che utilizza molte finestre, di cui una sola alla volta è attiva, può trarre vantaggio da un'implementazione come insieme di ULT su un singolo processo leggero. Il vantaggio di restringere tali applicazioni agli ULT è l'efficienza; gli ULT si possono creare, distruggere, bloccare, attivare, ecc. senza coinvolgere il kernel, mentre, se il kernel vedesse ogni ULT, dovrebbe allocare delle strutture dati per ognuno e gestire il cambio di thread. Come si è visto (Tabella 4.1), il cambio di thread a livello del kernel è più costoso rispetto al cambio di thread a livello utente.

Se un'applicazione contiene thread che possono bloccarsi, ad esempio effettuando I/O, allora avere molti processi leggeri a supporto di un numero maggiore o uguale di ULT è attraente; né l'applicazione né la libreria per i thread devono specificare algoritmi complicati per supportare l'esecuzione di altri thread all'interno dello stesso processo; se, invece, un thread di un processo si blocca, gli altri thread di quel processo possono entrare in esecuzione, grazie agli altri processi leggeri.

La mappatura uno-a-uno fra ULT e processi leggeri è efficace per alcune applicazioni, ad esempio, un calcolo parallelo su array potrebbe dividere le righe degli array fra diversi thread; se c'è esattamente un ULT per ogni processo leggero, allora non sarà necessario alcun cambio di thread durante il calcolo.

Un misto di thread permanentemente legati ai processi leggeri e thread non legati (thread multipli che condividono processi leggeri multipli) è appropriato per alcune applicazioni. Ad

esempio, una applicazione in tempo reale potrebbe richiedere che alcuni thread abbiano priorità di sistema e schedulazione in tempo reale, mentre altri effettuano funzioni in background e possono condividere uno o più processi leggeri.

Struttura dei processi

La Figura 4.15 confronta, in termini generali, la struttura dei processi di un sistema UNIX tradizionale con quella di Solaris. In una tipica implementazione di UNIX, la struttura dei processi comprende l'identificatore di processo, l'identificatore dell'utente, la tabella di allocazione dei segnali che il kernel usa per decidere cosa fare quando manda un segnale ad un processo, i file descriptor che descrivono lo stato dei file usati dal processo, una mappa di memoria che definisce lo spazio di indirizzamento del processo, e una struttura di stato del processore che comprende lo stack del kernel per il processo. Solaris mantiene questa struttura di base, ma rimpiazza lo stato del processore con una lista di strutture contenenti un blocco di dati per ogni processo leggero.

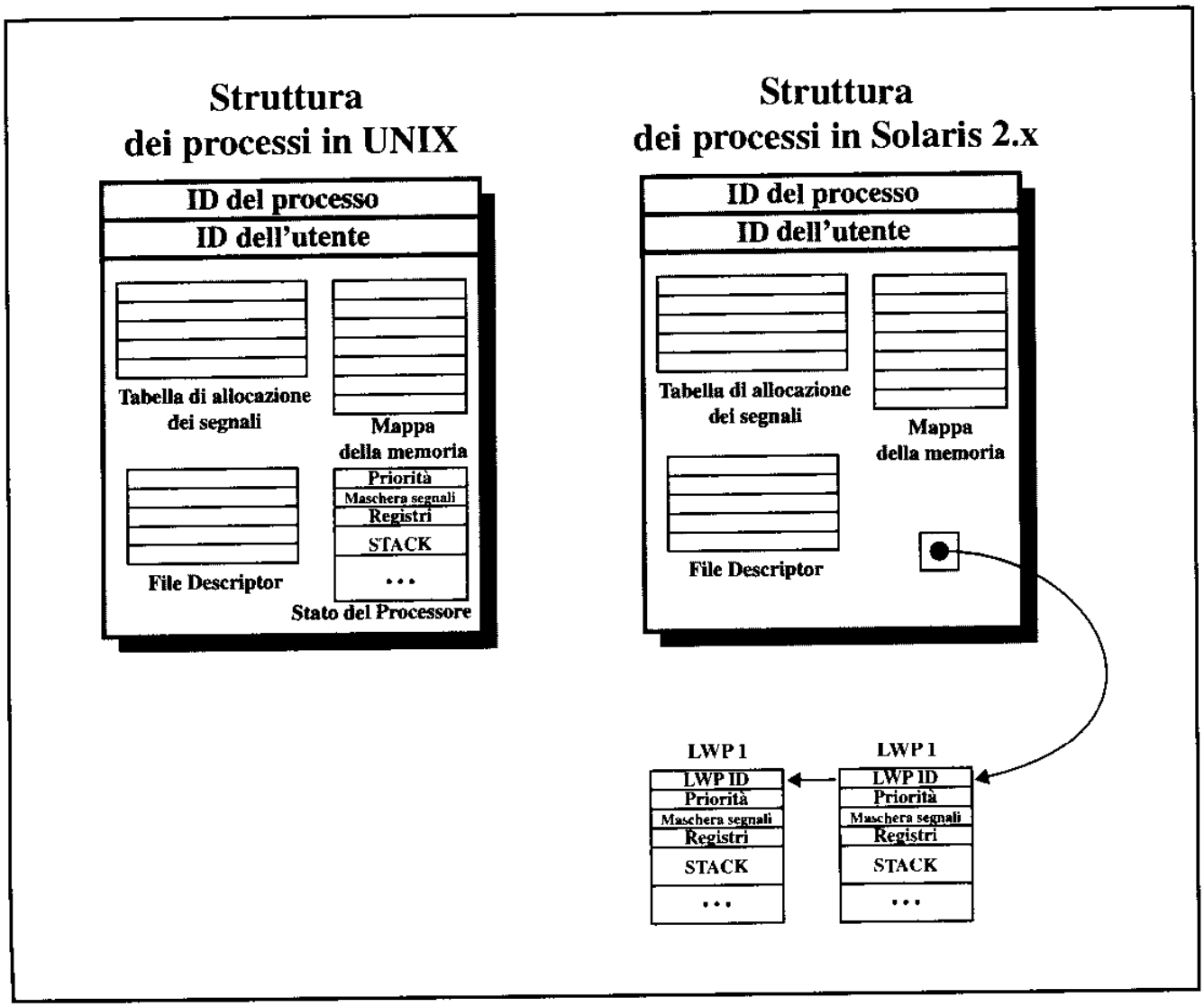


Figura 4.15 *Struttura dei processi in un sistema UNIX tradizionale e in Solaris 2.x [LEWI96]*

La struttura dati dei processi leggeri comprende i seguenti elementi:

- Un identificatore di processo leggero.
- La priorità di questo processo leggero e quindi il processo del kernel che lo supporta.
- Una maschera dei segnali che indica al kernel quali segnali saranno accettati.
- I valori salvati dei registri a livello utente (quando il processo leggero non è in esecuzione).
- Lo stack del kernel per questo processo leggero, che contiene gli argomenti delle chiamate di sistema, i risultati e i codici di errore per ogni livello di chiamata.
- L'utilizzo delle risorse e i dati di monitoraggio.
- Un puntatore al thread del kernel corrispondente.
- Un puntatore alla struttura del processo.

Esecuzione dei thread

La Figura 4.16 mostra una visione semplificata degli stati di esecuzione di ULT e processi leggeri. L'esecuzione di thread a livello utente è gestita dalla libreria per i thread. Si considerino prima i thread non limitati (cioè quelli che condividono alcuni processi leggeri). Un thread non limitato può essere in uno dei quattro stati: Runnable, Active, Sleeping o Stopped. Un ULT in stato Active è assegnato ad un processo leggero e viene eseguito contemporaneamente al thread del kernel. Vari eventi possono portare tale ULT a lasciare lo stato Active. Si consideri T1, un ULT in stato Active, a cui può accadere uno degli eventi seguenti:

- **Sincronizzazione:** T1 chiama una delle primitive per la concorrenza discusse nel Capitolo 5, per coordinare la propria attività con altri thread e per garantire la mutua esclusione; allora, T1 è posto in stato Sleeping, e quando la condizione di sincronizzazione è soddisfatta, T1 passa in stato Runnable.
- **Sospensione:** ogni thread (compreso T1) può sospendere T1 e farlo passare in stato Stopped; T1 rimane in quello stato finché un altro thread invia una richiesta di continuazione, che lo fa passare in stato Runnable.
- **Prelazione:** un thread attivo (T1 o qualche altro thread) fa qualcosa che fa passare un altro thread (T2) a priorità più alta in stato Runnable. Se T1 è il thread attivo con priorità più bassa, viene interrotto e mandato in stato Runnable, e T2 viene assegnato al processo leggero resosi disponibile.
- **Concessione:** se T1 esegue il comando di libreria *thr_yield()*, lo schedulatore della libreria cerca un altro thread eseguibile (T2) avente la stessa priorità. Se lo trova, T1 è messo in stato Runnable e T2 è assegnato al processo leggero libero; altrimenti, T1 continua l'esecuzione.

In tutti i casi precedenti, quando T1 viene spostato dallo stato Active, la libreria dei thread sceglie un altro thread libero in stato Runnable e lo esegue sul processo leggero resosi disponibile.

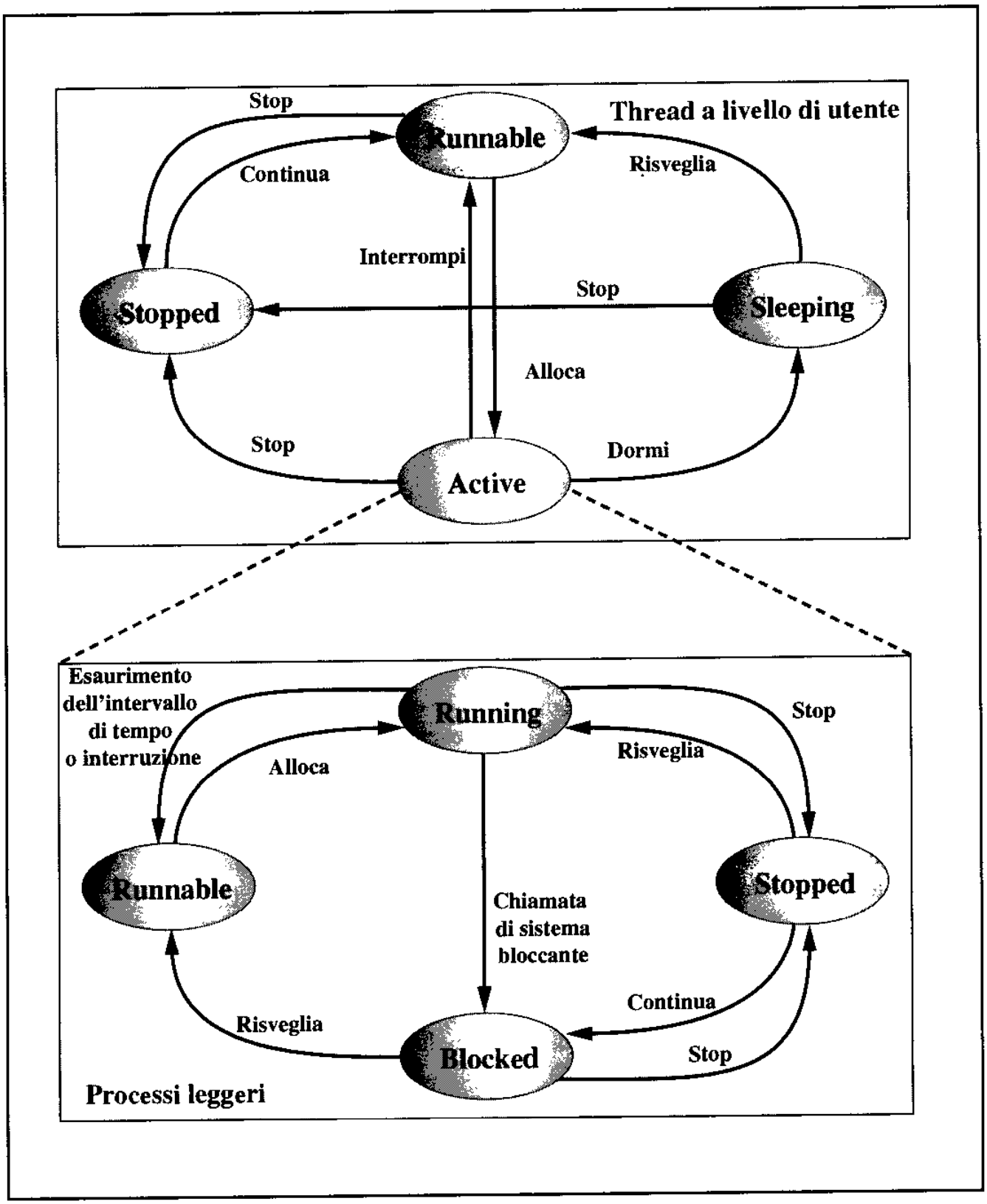


Figura 4.16 *Thread a livello utente e stati dei processi leggeri in Solaris*

La Figura 4.16 mostra anche il diagramma di stato per un processo leggero, che costituisce una descrizione dettagliata dello stato attivo dell'ULT, perché un thread libero ha un solo processo leggero assegnato quando si trova in stato Active. Il diagramma di stato del processo

leggero si spiega da solo: un thread in stato Active è in esecuzione quando il suo processo leggero è in stato Running, e quando un thread attivo esegue una chiamata di sistema bloccante, il processo leggero entra in stato Blocked. Comunque, l'ULT rimane legato a quel processo leggero, e per quanto riguarda la libreria dei thread, quell'ULT rimane attivo.

Con thread legati, la relazione fra ULT e processo leggero è leggermente diversa; ad esempio, se un ULT legato passa in stato Sleeping aspettando un evento di sincronizzazione, il suo processo leggero deve anch'esso interrompere l'esecuzione. Questo si realizza facendo bloccare il processo leggero su una variabile di sincronizzazione a livello del kernel.

Interrupt come thread

Molti sistemi operativi contengono due forme fondamentali di attività asincrona: processi e interrupt. I processi (o i thread) cooperano fra loro e gestiscono l'uso di strutture dati condivise tramite varie primitive, che garantiscono la mutua esclusione (un solo processo alla volta può eseguire un certo codice o accedere a certi dati) e sincronizzano l'esecuzione. Gli interrupt sono sincronizzati impedendo che possano verificarsi per un certo periodo di tempo. Solaris unifica i due concetti in un solo modello: i thread del kernel e il meccanismo per schedulare ed eseguire i thread del kernel. Per far ciò, gli interrupt vengono convertiti in thread del kernel.

Il motivo di questa conversione è di ridurre il sovraccarico: i gestori delle interruzioni spesso operano su strutture dati condivise con il resto del kernel, e di conseguenza, mentre una routine del kernel che accede quei dati è in esecuzione, gli interrupt devono essere bloccati, anche se molti di essi non modificherebbero quei dati. Tipicamente, per realizzare ciò la routine aumenta il livello di priorità dell'interrupt, in modo da bloccarli, e ristabilisce la priorità precedente dopo aver completato l'accesso ai dati. Queste operazioni richiedono tempo, e il problema si ingigantisce su un sistema multiprocessore: il kernel, per proteggere più oggetti, può dover bloccare gli interrupt su tutti i processori.

La soluzione di Solaris si può riassumere così:

1. Solaris impiega un insieme di thread per gestire gli interrupt: come gli altri thread del kernel, i thread per gli interrupt hanno i loro identificatori, priorità, contesti e stack.
2. Il kernel controlla l'accesso alle strutture dati e sincronizza i thread per gli interrupt utilizzando primitive per la mutua esclusione, del tipo discusso nel Capitolo 5; in altri termini, usa le normali tecniche di sincronizzazione per gestire gli interrupt.
3. I thread per gli interrupt hanno priorità più alta di tutti gli altri tipi di thread del kernel.

Quando si verifica un interrupt, viene mandato ad un particolare processore, e il thread correntemente in esecuzione su quel processore viene *congelato*. Un thread congelato non può essere spostato su un altro processore, e il suo contesto viene salvato: è semplicemente sospeso fino al termine dell'esecuzione dell'interrupt, mentre il processore inizia immediatamente ad eseguire il thread per l'interrupt. Infatti, esiste sempre un gruppo di thread disattivati, disponibili per gli interrupt, per cui non è necessario attendere di crearne uno nuovo. Il thread per l'interrupt quindi effettua le operazioni necessarie per la gestione dell'interrupt; se si deve accedere ad una

struttura dati che in qualche modo è bloccata da un altro thread in esecuzione, il thread per l'interrupt deve aspettare per potervi accedere. Un thread per gli interrupt può essere interrotto solo da un altro thread per interrupt con priorità più alta.

L'esperienza con i thread per interrupt di Solaris indica che questo approccio fornisce prestazioni superiori rispetto alla strategia normale di gestione degli interrupt [KLEI95].

4.6 Sommario

Alcuni sistemi operativi distinguono i concetti di processo e thread; il primo è associato al possesso di risorse e il secondo all'esecuzione di programmi. Questo approccio può portare ad un aumento di efficienza e ad un modello di programmazione più conveniente. In un sistema multithread, si possono avere molti thread concorrenti all'interno di ciascun processo. Ciò si può realizzare utilizzando thread a livello utente o thread a livello di kernel. I thread a livello utente non sono visti dal sistema operativo, e possono essere creati e gestiti da una libreria per i thread, che viene eseguita nello spazio utente di un processo. I thread a livello utente sono molto efficienti perché non è necessario effettuare un cambiamento di modalità per passare da un thread all'altro; però, è possibile eseguire un solo thread a livello utente alla volta per ogni processo, e se un thread si blocca, l'intero processo è bloccato. I thread a livello di kernel sono anche essi thread all'interno di un processo, ma vengono gestiti dal kernel; per questo motivo più thread dello stesso processo possono essere eseguiti in parallelo su un multiprocessore, e il blocco di un thread non comporta il blocco dell'intero processo. Purtroppo, è necessario effettuare un cambiamento di modalità per passare da un thread ad un altro.

Il multiprocessing simmetrico è un metodo per organizzare un sistema multiprocessore in modo che ogni processo (o thread), compresi anche il codice e i processi del kernel, possa essere eseguito su ciascun processore. Un'architettura a multiprocessore simmetrico pone nuovi problemi per la progettazione dei sistemi operativi, e, a parità di condizioni, fornisce prestazioni superiori rispetto ad un sistema monoprocessore.

Negli ultimi anni c'è stato molto interesse per l'approccio a microkernel alla la progettazione dei sistemi operativi. Nella sua forma pura, un sistema operativo a microkernel si compone di un microkernel molto piccolo che viene eseguito in modalità kernel, e che contiene solamente le funzioni essenziali e più critiche del sistema operativo; le altre funzioni del sistema operativo sono implementate in modo da essere eseguite in modalità utente, utilizzando il microkernel per i compiti critici. La progettazione a microkernel porta ad un'implementazione estremamente modulare e flessibile, ma in ogni caso le prestazioni di tale architettura rimangono un problema aperto.

4.7 Letture raccomandate

[LEWI96] e [KLEI96] forniscono una buona panoramica del concetto di thread e una discussione sulle strategie di programmazione; il primo è più orientato ai concetti e il secondo alla programmazione, ma entrambi danno una buona descrizione dei due aspetti. [PHAM96] tratta in dettaglio l'architettura dei thread di Windows NT.

[MUKH96] svolge una buona analisi della progettazione dei sistemi per multiprocessore simmetrico. [CHAP97] contiene cinque articoli sulle tendenze recenti nella progettazione di sistemi operativi multiprocessore. [LIED95] e [LIED96] discutono i principi di progettazione dei microkernel; il secondo ne analizza le prestazioni.

CHAP97 Chapin S. e Maccabe A. editor. "Multiprocessor Operating Systems: Harnessing the Power". Special issue of *IEEE Concurrency*, Aprile-Giugno 1997.

KLEI96 Kleiman S., Shah D. e Smallders B. *Programming with Threads*. Upper Saddle River, NJ: Prentice Hall, 1996.

LEWI96 Lewis B. e Berg D. *Threads Primer*. Upper Saddle River, NJ: Prentice Hall, 1996.

LIED95 Liedtke J. "On μ-Kernel Construction". *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dicembre 1995.

LIED96 Liedtke J. "Toward Real Microkernels". *Communications of the ACM*, Settembre 1996.

MUKH96 Mukherjee B. e Karsten S. "Operating Systems for Parallel Machines". In *Parallel Computers: Theory and Practice*, edito da T. Casavant, P. Tvrkik e F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

PHAM96 Pham T. e Garg P. *Multithreaded Programming with Windows NT*. Upper Saddle River, NJ: Prentice Hall, 1996.

4.8 Problemi

4.1 Si è visto che i due vantaggi dell'uso di thread multipli per ogni processo sono i seguenti: (1) la creazione di un nuovo thread all'interno di un processo richiede meno lavoro della creazione di un processo, e (2) la comunicazione fra thread dello stesso processo è semplificata. È vero che anche un cambiamento di contesto fra due thread dello stesso processo richiede meno lavoro rispetto ad un cambiamento di contesto fra due thread di processi diversi?

4.2 In OS/2, quello che normalmente in altri sistemi operativi è compreso nel concetto di processo, viene diviso in tre tipi di entità: sessioni, processi e thread. Una sessione è un insieme di processi associati ad un'interfaccia con l'utente (tastiera, video, mouse); le sessioni rappresentano quindi applicazioni interattive dell'utente, come un programma di elaborazione di testo o un foglio di calcolo. Questo concetto permette all'utente del per-

sonal computer di aprire più di un'applicazione, ciascuna con una o più finestre sullo schermo. Il sistema operativo deve tenere traccia della finestra attiva (quindi della relativa sessione), così l'input di tastiera e mouse è diretto verso tale sessione. In ogni istante, una sessione è in foreground e le altre sono in background, e tutto l'input della tastiera e del mouse è diretto verso uno dei processi della sessione attiva, come stabilito dall'applicazione. Quando una sessione è in foreground un processo che effettua output su video manda i comandi direttamente al buffer hardware del video, e quindi allo schermo dell'utente; quando la sessione viene messa in background, il buffer hardware del video è salvato in un buffer logico del video per quella sessione, in modo che, se un thread di un processo di una sessione in background produce output per lo schermo, questo venga diretto al buffer logico del video. Quando la sessione ritorna in foreground, lo schermo viene aggiornato mostrando il contenuto del buffer logico del video della sessione. C'è un modo semplice per ridurre il numero di concetti relativi ai processi in OS/2 da tre a due: eliminare le sessioni e associare l'interfaccia con l'utente (tastiera, mouse, schermo) ai processi. Così si ha un solo processo in foreground alla volta. Per aumentare la strutturazione, i processi possono essere divisi in thread.

a. Quali benefici vengono persi con questo approccio?

b. Se si procede con queste modifiche, dove è meglio assegnare le risorse (memoria, file, ecc.): ai processi o ai thread?

4.3 Si consideri un ambiente in cui ci sia una mappatura uno-a-uno fra thread a livello utente e thread a livello del kernel, e che permetta a uno o più thread di un processo di effettuare chiamate di sistema bloccanti mentre gli altri thread continuano l'esecuzione. Spiegare perché, in questo modello, i programmi multithread sono più efficienti rispetto a programmi equivalenti, composti da un solo thread, eseguiti su una macchina uniprocessore.

4.4 Se un processo termina, cosa accade ai suoi thread ancora in esecuzione?

4.5 Il sistema operativo OS/390 per mainframe è strutturato intorno ai concetti di *spazio di indirizzamento* e *task*. Semplificando, uno spazio di indirizzamento corrisponde ad un'applicazione, e in altri sistemi operativi corrisponde più o meno a un processo. All'interno di uno spazio di indirizzamento, è possibile generare vari processi che vengono eseguiti concorrentemente; ciò corrisponde al concetto di multithreading. Ci sono due strutture dati fondamentali per gestire questa struttura di processi. Un blocco di controllo dello spazio di indirizzamento (ASCB, *address space control block*) contiene le informazioni su uno spazio di indirizzamento necessarie a OS/390, indipendentemente dal fatto che lo spazio di indirizzamento possa essere in esecuzione; fra tali informazioni vi sono: la priorità di allocazione, la memoria reale e virtuale allocata ad esso, il numero di processi pronti per l'esecuzione in questo spazio di indirizzamento, e se esso sia stato scaricato sul disco. Un blocco di controllo del task (TCB, *task control block*) rappresenta un programma utente in esecuzione: contiene informazioni necessarie per gestire un task in uno spazio di indirizzamento, tra cui quelle sullo stato del processore, i puntatori a programmi che fanno parte di questo task e lo stato di esecuzione del task. Gli ASCB sono strutture globali mantenute nella memoria di sistema, mentre i TCB sono strutture locali mantenu-

te all'interno del loro spazio di indirizzamento. Qual è il vantaggio di separare le informazioni di controllo tra strutture locali e globali?

4.6 Un multiprocessore con otto processori ha 20 lettori di nastri; sono stati inviati al sistema molti job, ciascuno dei quali richiede al massimo quattro lettori di nastri per completare l'esecuzione. Si supponga che ogni job inizi l'esecuzione con solamente tre lettori, e che prosegua per molto tempo prima di richiedere il quarto lettore, usandolo per breve tempo verso la fine dell'esecuzione; si supponga inoltre che tali job continuino ad arrivare senza sosta.

a. Se lo schedulatore del sistema operativo non attiva un job finché non ci sono quattro lettori disponibili, e, quando un job viene avviato, gli assegna immediatamente quattro lettori, rilasciati solo al termine dell'esecuzione del job, allora qual è il numero massimo di job che possono procedere contemporaneamente? Qual è il numero massimo e minimo di lettori che possono essere lasciati inutilizzati con questa politica?

b. Si suggerisca una politica alternativa per migliorare l'utilizzazione dei lettori e nello stesso tempo evitare possibili stalli del sistema. Qual è il massimo numero di job che possono procedere contemporaneamente? Qual è il numero massimo e il minimo di lettori inutilizzati?

4.7 Nella descrizione degli stati degli ULT in Solaris, si è affermato che un ULT può lasciare il posto ad un altro thread con la stessa priorità. È possibile che ci sia un thread in stato Runnable con priorità più alta, e che quindi la funzione di concessione abbia come risultato l'attivazione di un thread con priorità uguale o maggiore?

C A P I T O L O 5

CONCORRENZA: MUTUA ESCLUSIONE E SINCRONIZZAZIONE

I temi centrali della progettazione di un sistema operativo sono connessi con la gestione di processi e thread:

- **Multiprogrammazione:** gestione di processi multipli in un sistema a singolo processore. In pratica tutti i sistemi operativi recenti forniscono la multiprogrammazione.
- **Multiprocessing:** gestione di processi multipli in un multiprocessore. Fino a poco tempo fa, l'organizzazione a multiprocessing era utilizzata solo per sistemi di grandi dimensioni, come mainframe e minicomputer di fascia alta; più recentemente, il multiprocessing è apparso nei server e nelle workstation.
- **Processi distribuiti:** gestione di processi multipli che sono eseguiti su sistemi distribuiti: lo sviluppo recente dei cluster è l'esempio principale di questo tipo di sistemi.

Un concetto fondamentale, per tutti questi temi e per la progettazione dei sistemi operativi, è quello di concorrenza. La concorrenza comprende diversi aspetti di progettazione, fra cui la comunicazione fra processi, la condivisione e la competizione per le risorse, la sincronizzazione delle attività di processi multipli e l'allocazione di tempo di processore ai processi. Si vedrà che questi elementi non sono presenti solo in ambienti multiprocessing e distribuiti, ma anche in sistemi in multiprogrammazione con un solo processore.

La concorrenza appare in tre contesti diversi:

- **Applicazioni multiple:** la multiprogrammazione fu inventata per permettere di dividere dinamicamente il tempo di calcolo fra le applicazioni attive.
- **Applicazioni strutturate:** come estensione dei principi di progettazione modulare e programmazione strutturata, alcune applicazioni possono essere programmate efficacemente come insiemi di processi concorrenti.
- **Struttura del sistema operativo:** anche il programmatore di sistemi ottiene gli stessi vantaggi dalla strutturazione, e si è visto che, spesso, anche i sistemi operativi sono implementati come insiemi di processi.

Data l'importanza dell'argomento, quattro capitoli di questo libro trattano concetti relativi alla concorrenza: questo capitolo e il prossimo esaminano la concorrenza in sistemi multiprogrammati e multiprocessing; i Capitoli 13 e 14 esaminano i concetti di concorrenza relativi al calcolo distribuito. Sebbene il resto del libro tratti altri argomenti interessanti legati alla progettazione di sistemi operativi, l'aspetto della concorrenza è prioritario nella trattazione degli altri argomenti.

Questo capitolo inizia con un'introduzione al concetto di concorrenza e studiando poi le implicazioni dell'esecuzione di diversi processi concorrenti[1]. Il requisito basilare per il supporto dei processi concorrenti è la capacità di garantire la mutua esclusione, cioè l'abilità di escludere tutti gli altri processi da un'azione, mentre un solo processo è abilitato ad eseguirla. Nella seconda sezione di questo capitolo si esaminano alcuni approcci per ottenere la mutua esclusione: tutti gli approcci sono soluzioni software, e richiedono l'uso di una tecnica chiamata *busy waiting*. Nel seguito saranno esaminati alcuni meccanismi hardware per la mutua esclusione; quindi, si vedranno soluzioni che non impiegano il busy waiting e che possono essere fornite dal sistema operativo o garantite dal compilatore. Si esamineranno tre approcci: semafori, monitor e scambio di messaggi.

Si useranno due problemi classici di concorrenza per illustrare i concetti e paragonare gli approcci presentati in questo capitolo. Il problema del produttore/consumatore è introdotto all'inizio ed è usato in seguito come esempio; il capitolo si conclude con il problema dei lettori/scrittori.

La trattazione della concorrenza continua nel Capitolo 6, e la discussione sui meccanismi per la concorrenza nei sistemi reali è rimandata alla fine di quel capitolo.

5.1 Principi della concorrenza

In un sistema a singolo processore con multiprogrammazione, i processi sono alternati nel tempo per dare l'illusione dell'esecuzione simultanea (vedi Figura 2.12a). Anche se il parallelismo non è reale, e c'è un sovraccarico dovuto agli scambi di contesto fra processi, l'esecuzione alternata porta grandi benefici dal punto di vista dell'efficienza di esecuzione e della strutturazione

[1] Per semplicità, in generale si farà riferimento all'esecuzione concorrente di *processi*. Infatti, come si è visto nel capitolo precedente, in alcuni sistemi l'unità fondamentale di concorrenza è il thread, anziché il processo.

dei programmi. In un sistema multiprocessore è possibile non solo alternare le esecuzioni dei processi ma anche sovrapporle (Figura 2.12b).

A prima vista sembrerebbe che l'alternanza e la sovrapposizione rappresentino modi di esecuzione fondamentalmente diversi, e che presentino problemi diversi; in realtà entrambe le tecniche si possono vedere come esempi di elaborazione concorrente e presentano gli stessi problemi. Nel caso di un singolo processore, i problemi sorgono da una caratteristica dei sistemi multiprogrammati: non è possibile predire le velocità relative di esecuzione dei processi, poiché dipendono dalle attività degli altri processi, dal modo in cui il sistema operativo gestisce gli interrupt e dalla politica di schedulazione del sistema. Si presentano queste difficoltà:

1. La condivisione di risorse globali è pericolosa: ad esempio, se due processi fanno entrambi uso della stessa variabile globale ed effettuano letture e scritture su quella variabile, allora l'ordine in cui le operazioni di lettura e scrittura sono eseguite è critico. Un esempio di questo problema è trattato nella sottosezione seguente.
2. Per il sistema operativo è difficile gestire l'assegnazione di risorse in maniera ottimale. Ad esempio, un processo A può richiedere l'uso di un particolare canale di I/O ed essere sospeso prima di poterlo usare. Per il sistema operativo, bloccare semplicemente il canale e impedirne l'uso da parte degli altri processi può essere inefficiente.
3. Trovare un errore di programmazione diventa molto difficile, perché i risultati tipicamente non sono riproducibili (si vedano ad es. [LEBL87] e [CARR89] per una discussione su questo punto).

Tutte queste difficoltà si presentano anche in un sistema multiprocessore, perché anche in quel caso la velocità di esecuzione relativa non si può predire. Un sistema multiprocessore deve anche gestire problemi dovuti all'esecuzione simultanea dei processi. Fondamentalmente, comunque i problemi sono gli stessi incontrati nei sistemi a singolo processore; e questo sarà chiarito nel seguito.

Un semplice esempio

Si consideri la seguente procedura:

```
procedure echo;
var out, in: character;
begin
   input(in,tastiera);
   out := in;
   output(out, schermo);
end.
```

Questa procedura mostra gli elementi essenziali di un programma che realizza l'eco, in altre parole copia l'input nell'output; l'input è letto dalla tastiera, un tasto alla volta. Ogni carattere di input è memorizzato nella variabile *in*, poi è copiato nella variabile *out* e mandato allo schermo.

Ogni programma può chiamare questa procedura ripetutamente per leggere l'input dell'utente e visualizzarlo sul suo schermo.

Si consideri il caso di un sistema multiprogrammato con un solo processore ed un unico utente, che può passare da un'applicazione a un'altra: ogni applicazione usa la stessa tastiera per l'input e lo stesso schermo per l'output. Poiché ogni applicazione deve usare la procedura *echo*, ha senso che tale procedura sia condivisa e sia caricata in una porzione di memoria globale, accessibile da tutte le applicazioni; in tal modo, si usa una sola copia della procedura, risparmiando spazio.

La condivisione di memoria centrale fra i processi è utile per avere interazioni efficienti; comunque questa condivisione può creare dei problemi. Si consideri questa sequenza:

1. Il processo P1 chiama la procedura *echo* ed è interrotto subito dopo la conclusione della lettura dell'input. A questo punto l'ultimo carattere digitato, x, è memorizzato nella variabile *in*.
2. Il processo P2 è attivato e chiama la procedura *echo*, che è eseguita fino alla fine: legge un carattere y e lo stampa sullo schermo.
3. Il processo P1 è riattivato, e a questo punto la variabile *in* è stata sovrascritta e il valore x è andato perso, sostituito da y, che è copiato in *out* e stampato sullo schermo.

Così il primo carattere è perso e il secondo è stampato due volte: la causa del problema è la variabile condivisa *in*. Tutti i processi hanno accesso a questa variabile, e se un processo la sovrascrive e poi è interrotto, un altro processo ne può alterare il valore prima che il primo processo possa utilizzarlo. Si supponga invece che un solo processo alla volta possa eseguire la procedura; allora la precedente sequenza sarebbe sostituita da questa:

1. Il processo P1 chiama la procedura *echo* ed è interrotto subito dopo la lettura dell'input. A questo punto l'ultimo carattere digitato, x, è memorizzato nella variabile *in*.
2. Il processo P2 è attivato e chiama la procedura *echo*, ma, poiché P1 è stato sospeso mentre stava eseguendo la procedura, P2 è bloccato e non può accedere alla procedura: P2 si sospende nell'attesa che *echo* diventi disponibile.
3. Più tardi, il processo P1 è riattivato e completa l'esecuzione di *echo*; è stampato correttamente il carattere x.
4. Quando P1 esce da *echo*, il blocco di P2 è rimosso, e quando poi P2 è riattivato, riesce a chiamare la procedura *echo*.

La lezione da imparare da quest'esempio è che è necessario proteggere le variabili globali condivise (e altre risorse globali condivise), e che l'unico modo per farlo è controllare il codice che ha accesso alle variabili. Se s'impone la regola che un solo processo alla volta può eseguire *echo*, e che una volta dentro la procedura è necessario completarne l'esecuzione prima di dare l'accesso ad un altro processo, allora il tipo di errore che abbiamo visto non può verificarsi. L'argomento principale di questo capitolo è come garantire che ciò avvenga.

Nel descrivere il problema si è formulata l'ipotesi che ci sia un solo processore e che il

sistema operativo fornisca la multiprogrammazione: l'esempio dimostra che i problemi legati alla concorrenza sorgono anche in presenza di un solo processore. In un sistema multiprocessore c'è lo stesso problema di protezione delle risorse condivise e valgono le stesse soluzioni. Per prima cosa si supponga che non ci sia nessun meccanismo per controllare l'accesso alla variabile globale condivisa:

1. I processi P1 e P2 sono entrambi in esecuzione, ciascuno su un processore diverso, ed entrambi chiamano la procedura *echo*.
2. Si verificano i seguenti eventi, dove due eventi sulla stessa linea accadono contemporaneamente:

```
Processo P1                    Processo P2
...                            ...
input(in,tastiera);            ...
...                            input(in,tastiera);.
out := in;                     out := in;
output(out, schermo);          ...
...                            output(out, schermo);
...                            ...
```

Il risultato è che il carattere letto da P1 è perso prima di essere visualizzato, e il carattere letto da P2 è stampato sia da P1 sia da P2. Come prima, si supponga di poter garantire che un solo processo alla volta esegua *echo*; allora si ha questa sequenza:

1. I processi P1 e P2 sono entrambi in esecuzione, ciascuno su un processore diverso. P1 chiama la procedura *echo*.
2. Mentre P1 sta eseguendo la procedura *echo*, anche P2 la chiama; ma poiché P1 è ancora all'interno di *echo* (sia che P1 fosse in esecuzione, sia che fosse sospeso), P2 non può entrare nella procedura, quindi è sospeso nell'attesa che la procedura sia disponibile.
3. Più tardi il processo P1 completa l'esecuzione di *echo*, esce dalla procedura e continua l'esecuzione; immediatamente dopo l'uscita di P1, P2 è riattivato e inizia ad eseguire *echo*.

Nel caso di un sistema a singolo processore, la ragione del problema è che un'interruzione può fermare l'esecuzione di un processo in qualunque momento. Nel caso di un sistema multiprocessore, oltre a questo c'è il problema causato da due processi che sono eseguiti simultaneamente e cercano entrambi di accedere alla stessa variabile globale. Comunque, la soluzione ad entrambi i tipi di problema è la stessa: controllare l'accesso alle risorse condivise.

Problemi di concorrenza nei sistemi operativi

Quali problemi di progettazione e di gestione nascono a causa della concorrenza? Ci sono i seguenti punti da tenere in considerazione:

1. Il sistema operativo deve poter tenere traccia dei processi attivi: ciò si realizza utilizzando blocchi di controllo dei processi, come descritto nel Capitolo 4.

2. Il sistema operativo deve allocare e deallocare varie risorse per ogni processo attivo, tra cui:
   - Tempo di elaborazione: gestito dallo schedulatore, discusso nella Parte Quarta.
   - Memoria: molti sistemi operativi utilizzano un meccanismo di memoria virtuale; l'argomento è oggetto della Parte Terza.
   - File: sono discussi nel Capitolo 12.
   - Dispositivi di I/O: sono discussi nel Capitolo 11.
3. Il sistema operativo deve proteggere i dati e le risorse fisiche di ogni processo da interferenze involontarie da parte di altri processi: ciò richiede tecniche che hanno effetto su memoria, file e dispositivi di I/O. Una trattazione generale della protezione si trova nel Capitolo 15.
4. Il risultato di un processo deve essere indipendente dalla sua velocità di esecuzione, relativamente a quella degli altri processi concorrenti: questo è l'argomento del presente capitolo.

Per capire come si può ottenere l'indipendenza dalla velocità di esecuzione, è necessario esaminare come avviene l'interazione fra processi.

Interazione fra processi

Si può classificare il modo in cui i processi interagiscono sulla base del grado di conoscenza che hanno dell'esistenza degli altri processi. La Tabella 5.1 elenca i tre possibili gradi di conoscenza con le conseguenze di ciascuno:

- **Processi che non si vedono tra loro:** sono processi indipendenti che non sono fatti per collaborare: questa situazione si verifica con la multiprogrammazione di processi indipendenti. Essi possono essere o programmi di calcolo puro, o sessioni interattive, o un misto: sebbene tali processi non comunichino fra loro, il sistema operativo deve preoccuparsi della gestione della **competizione** per le risorse. Ad esempio due applicazioni indipendenti possono richiedere entrambe l'accesso allo stesso disco, file o stampante; il sistema operativo deve regolare gli accessi.
- **Processi che vedono gli altri processi indirettamente:** sono processi che non conoscono necessariamente il nome degli altri processi, ma condividono con loro l'accesso a qualche oggetto, come un buffer di I/O. Tali processi effettuano **cooperazione** nel senso che condividono un oggetto comune.
- **Processi che vedono gli altri processi direttamente:** sono processi che possono comunicare direttamente fra loro per nome, e che sono progettati per lavorare insieme; anche questi effettuano **cooperazione**.

Le condizioni non saranno mai ben definite come nella Tabella 5.1, ma in generale si avranno sia cooperazione sia competizione fra processi. Comunque è utile esaminare separatamente i tre casi della lista precedente per vedere le implicazioni che hanno per il sistema operativo.

Tabella 5.1 *Interazione fra processi*

Grado di conoscenza	Relazione	Influenza che un processo ha sugli altri	Possibili problemi di controllo
Processi che non si vedono tra loro	Competizione	• I risultati di un processo non dipendono dalle informazioni ottenute dagli altri • Il tempo di esecuzione dei processi può cambiare	• Mutua esclusione • Stallo (risorse riutilizzabili) • Starvation
Processi che vedono gli altri processi indirettamente (es. oggetti condivisi)	Cooperazione tramite condivisione	• I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri • Il tempo di esecuzione dei processi può cambiare	• Mutua esclusione • Stallo (risorse riutilizzabili) • Starvation • Coerenza dei dati
Processi che vedono gli altri processi direttamente (usano primitive di comunicazione)	Cooperazione tramite comunicazione	• I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri • Il tempo di esecuzione dei processi può cambiare	• Stallo (risorse non riutilizzabili) • Starvation

Competizione fra processi per le risorse

I processi concorrenti entrano in conflitto quando competono per l'uso della stessa risorsa. La situazione nella sua forma pura si può descrivere così: due o più processi vogliono accedere ad una risorsa durante la loro esecuzione. Ogni processo non sa dell'esistenza degli altri, ed ognuno deve procedere senza essere influenzato dall'esecuzione degli altri; ne deriva che ogni processo deve lasciare inalterato lo stato delle risorse dopo l'uso. Sono esempi di risorse i dispositivi di I/O, la memoria, il tempo del processore e il clock.

Fra i processi in competizione non c'è scambio di informazioni, ma l'esecuzione di un processo può influenzare il comportamento degli altri: in particolare, se due processi vogliono accedere alla stessa risorsa, uno dei due otterrà la risorsa dal sistema operativo e l'altro dovrà aspettare, rallentando la propria azione. In un caso limite, il processo bloccato non avrà mai accesso alla risorsa e quindi non terminerà mai.

Nel caso di competizione fra processi bisogna affrontare tre problemi di controllo: il primo problema è la **mutua esclusione**. Si supponga che due o più processi richiedano di accedere ad una risorsa unica, che non può essere condivisa, come una stampante: durante l'esecuzione ogni processo manderà dei comandi al dispositivo di I/O, riceverà informazioni di stato, manderà

dati, e/o riceverà dati. Si farà riferimento alla risorsa in questione come *risorsa critica*, e la porzione di programma che ne fa uso sarà chiamata *sezione critica* del programma. È importante che un solo programma alla volta possa essere nella sezione critica: non si può semplicemente fare affidamento sul sistema operativo per capire e garantire questa restrizione, perché i dettagli possono essere non banali. Nel caso della stampante, ad esempio, si vuole che un solo processo abbia il controllo della stampante durante la stampa di un file, altrimenti sarà stampata un'alternanza di righe provenienti dai vari processi in competizione.

Una volta che si è garantita la mutua esclusione nascono altri due problemi di controllo; uno è il **deadlock** (stallo). Ad esempio, si considerino due processi P1 e P2, e due risorse R1 e R2, e si supponga che entrambi i processi abbiano bisogno di accedere ad entrambe le risorse per espletare parte dei loro compiti. Si potrebbe creare la seguente situazione: il sistema operativo assegna R1 a P2, e R2 a P1, tutti e due i processi aspettano l'altra risorsa, ma nessuno dei due rilascia la risorsa che possiede finché non ha acquisito l'altra risorsa ed effettuato il proprio compito. I due processi sono in stallo.

L'ultimo problema di controllo è la **starvation** (morte per fame). Si supponga che tre processi, P1, P2 e P3, richiedano l'accesso alla risorsa R, e si consideri la situazione in cui P1 è in possesso della risorsa, e sia P2 sia P3 sono in attesa di avere accesso alla risorsa. Quando P1 esce dalla sua sezione critica, riceverà l'accesso uno fra P2 e P3. Si supponga che questo sia P3, e che P1 chieda nuovamente l'accesso prima che P3 sia uscito dalla sezione critica. Se P1 riceve l'accesso dopo che P3 ha finito, e se P1 e P3 continuano a ricevere l'accesso alternativamente, allora a P2 sarà negato l'accesso indefinitamente, anche se non c'è stallo.

Il controllo della competizione coinvolge inevitabilmente il sistema operativo, perché è quello che alloca le risorse; inoltre gli stessi processi devono poter richiedere la mutua esclusione in qualche modo, ad esempio riservando una risorsa prima dell'uso. Qualsiasi soluzione dovrà fare uso del sistema operativo, ad esempio per fornire un meccanismo per prenotare le risorse. La Figura 5.1 mostra in modo astratto il meccanismo della mutua esclusione. Ci sono *n* processi che devono essere eseguiti concorrentemente. Ogni processo ha una sezione critica che opera su una risorsa R, e il resto del programma non fa uso di R. Ci sono due funzioni per garantire la mutua esclusione: *entracritica* e *escicritica*. Ogni funzione prende come argomento il nome della risorsa su cui si vuole garantire la mutua esclusione. Tutti i processi che tentano di entrare nella propria sezione critica, mentre un altro processo è nella propria sezione critica per la stessa risorsa, sono messi in attesa.

Rimangono da esaminare i meccanismi per costruire le funzioni *entracritica* e *escicritica*, che per il momento sono tralasciati per passare alla descrizione degli altri casi di interazione fra processi.

Cooperazione fra processi tramite condivisione

La cooperazione tramite condivisione riguarda processi che interagiscono fra loro senza vedere esplicitamente gli altri. Ad esempio, è possibile che molti processi abbiano accesso a variabili condivise, file, o basi di dati. I processi possono usare e modificare i dati condivisi senza fare riferimento agli altri processi, ma sapendo che altri processi possono accedere agli stessi dati; in definitiva, i processi devono cooperare per fare in modo che i dati che condividono siano gestiti correttamente. I meccanismi di controllo devono garantire l'integrità dei dati condivisi.

```
program mutuaesclusione;
const n = ...; (* numero di processi *)
procedure P(i: integer);
begin
  repeat
    entracritica(R);
    <sezione critica>;
    escicritica(R);
    <resto del programma>
  forever
end;
begin (* programma principale *)
  parbegin
    P(1);
    P(2);
    ...
    P(n);
  parend
end.
```

Figura 5.1 *Mutua esclusione*

Poiché i dati sono custoditi nelle risorse (dispositivi, memoria), i problemi di mutua esclusione, stallo e starvation si manifestano anche in questo caso; l'unica differenza è che ci sono due modi di accedere ai dati: lettura e scrittura, e solo le operazioni di scrittura devono essere mutuamente esclusive.

Comunque, oltre a questi problemi c'è anche una nuova proprietà che deve essere garantita, quella della coerenza dei dati. Un semplice esempio è quello di un'applicazione gestionale in cui vari dati possono essere aggiornati. Si supponga che due dati a e b debbano essere mantenuti nella relazione $a = b$, cioè che ogni programma che modifica uno dei due valori debba anche modificare l'altro per mantenere la relazione. Si considerino i due processi seguenti:

```
P1:  a := a + 1;
     b := b + 1;
P2:  b := 2 * b;
     a := 2 * a;
```

Se inizialmente lo stato è coerente, ogni processo preso separatamente lascerà i dati in uno stato coerente. Si consideri ora la seguente esecuzione concorrente, in cui i due processi rispettano la mutua esclusione su ognuno dei due dati (a e b):

```
a: = a + 1;
b: = 2 * b;
b: = b + 1;
a: = 2 * a;
```

Al termine dell'esecuzione di questa sequenza, la condizione $a = b$ non vale più; il problema si può evitare dichiarando sezione critica l'intera sequenza in entrambi i processi, anche se in realtà non c'è nessuna risorsa critica.

Vediamo così che il concetto di sezione critica è importante nel caso di cooperazione tramite condivisione: anche in questo caso si possono usare le funzioni astratte *entracritica* e *escicritica* viste prima (Figura 5.1). Questa volta, l'argomento delle funzioni potrebbe essere una variabile, un file, o qualunque altro oggetto condiviso. Inoltre, se le sezioni critiche sono usate per garantire l'integrità dei dati, è possibile che non si riescano a identificare delle risorse o variabili specifiche da passare come argomento; in tal caso si può usare come argomento un identificatore condiviso fra i processi concorrenti, che indichi le sezioni critiche che devono essere mutuamente esclusive.

Cooperazione fra processi tramite comunicazione

Nei primi due casi che sono stati discussi, ogni processo aveva il proprio ambiente, che non conteneva gli altri processi; l'interazione fra i processi era indiretta, e in entrambi i casi c'era condivisione. Nel caso di competizione, i processi condividono delle risorse senza vedere gli altri processi; nel secondo caso condividono dei valori, e sebbene ogni processo non veda esplicitamente gli altri, sa che è necessario mantenere l'integrità dei dati. Quando i processi cooperano tramite comunicazione, comunque, i processi sono tutti collegati fra di loro, per uno scopo comune; la comunicazione è un meccanismo per sincronizzare, o coordinare, le varie attività.

Tipicamente la comunicazione può essere caratterizzata dalla presenza di messaggi di qualche tipo. Le primitive per mandare e ricevere messaggi possono fare parte del linguaggio di programmazione, oppure essere fornite dal kernel del sistema operativo.

Poiché nell'atto di passare messaggi non c'è condivisione, per questo tipo di cooperazione non è necessario garantire la mutua esclusione; comunque ci sono ancora i problemi di stallo e starvation. Un esempio di stallo si ha quando due processi sono bloccati, ciascuno in attesa di una comunicazione da parte dell'altro. Si ha invece starvation quando, ad esempio, ci sono tre processi, P1, P2 e P3, che si comportano in questo modo: P1 tenta in continuazione di comunicare con P2 o P3, e sia P2 sia P3 tentano di comunicare con P1. Si può avere la sequenza in cui P1 e P2 continuano a scambiarsi informazioni, mentre P3 è bloccato in attesa di comunicare con P1. Non c'è stallo, perché P1 rimane attivo, ma P3 aspetta indefinitamente.

Requisiti per la mutua esclusione

Ogni meccanismo per fornire il supporto alla mutua esclusione deve avere questi requisiti:

1. La mutua esclusione deve essere garantita: fra i processi che hanno una sezione critica per lo stesso oggetto o risorsa condivisa, un solo processo alla volta può essere nella sezione critica.
2. Un processo che si ferma mentre è fuori dalla sezione critica non deve interferire con gli altri processi.

3. Non deve essere possibile che un processo che chiede l'accesso alla sezione critica sia fatto aspettare indefinitamente: non ci devono essere né stalli né starvation.
4. Quando nessun processo è nella sezione critica, a ogni processo deve essere concesso di entrare nella sua sezione critica senza attese.
5. Non si fanno supposizioni sulla velocità relativa dei processi, o sul numero di processori.
6. Ogni processo può rimanere nella sua sezione critica solo per un tempo finito.

Ci sono vari modi per soddisfare i requisiti per la mutua esclusione. Un modo è di scaricare la responsabilità ai processi che sono eseguiti concorrentemente; in tal modo i processi, siano essi programmi di sistema o applicazioni, dovrebbero coordinarsi fra loro per garantire la mutua esclusione, senza alcun supporto da parte del linguaggio di programmazione o del sistema operativo; tali approcci sono definiti approcci software. Sebbene questi approcci portino spesso ad un aumento del tempo di esecuzione e a commettere errori, è comunque utile analizzarli per capire meglio la complessità del calcolo concorrente: quest'argomento è trattato nella Sezione 5.2. Un secondo approccio richiede l'uso di istruzioni macchina particolari, che hanno il vantaggio di ridurre il sovraccarico, ma nella Sezione 5.3, dove sono descritte, si vedrà che non sono soddisfacenti. Un terzo approccio è quello in cui il supporto è dato dal sistema operativo o dal linguaggio di programmazione: i tre costrutti più importanti sono esaminati nelle Sezioni 5.4, 5.5 e 5.6.

5.2 Mutua esclusione: approcci software

Nel caso di processi concorrenti che sono eseguiti su un solo processore o su una macchina multiprocessore con memoria condivisa, si possono adottare degli approcci software. Solitamente si suppone di avere mutua esclusione a livello di accesso alla memoria ([LAMP91], ma si veda il Problema 5.10), cioè che gli accessi simultanei (lettura e/o scrittura) alla stessa locazione di memoria siano eseguiti in sequenza tramite qualche meccanismo di arbitraggio della memoria, anche se l'ordine di accesso non è specificato a priori. Non s'ipotizza nessun altro supporto dall'hardware, dal sistema operativo, o dal linguaggio di programmazione.

L'algoritmo di Dekker

Dijkstra [DIJK65] descrive un algoritmo di mutua esclusione per due processi, ideato dal matematico olandese Dekker. Seguendo l'approccio di Dijkstra, la soluzione è formulata per gradi; ciò ha il vantaggio di illustrare gli errori più comuni commessi durante lo sviluppo di programmi concorrenti. Durante lo sviluppo dell'algoritmo si useranno delle illustrazioni prese da [BEN82].

Primo tentativo

Come detto in precedenza, qualunque approccio alla mutua esclusione deve basarsi su qual-

che meccanismo fondamentale di esclusione a livello hardware; il più comune è il vincolo di effettuare un solo accesso alla volta ad una locazione di memoria. La Figura 5.2 mostra il "protocollo dell'iglù" come metafora di tale arbitraggio: l'entrata e l'iglù stesso sono talmente piccoli che una sola persona alla volta può essere all'interno, dove c'è una lavagna su cui si può scrivere un unico valore.

Un processo (P0 o P1) che vuole eseguire la sua sezione critica entra nell'iglù e guarda la lavagna; se c'è scritto il suo numero può uscire dall'iglù ed eseguire la sua sezione critica; altrimenti, esce dall'iglù e deve aspettare. Di tanto in tanto il processo entra nuovamente nell'iglù per controllare la lavagna, finché non arriva il momento in cui può entrare nella sua sezione critica. Questo processo è detto "busy waiting" (*attesa attiva*), perché il processo non può fare niente di produttivo finché non ha il permesso di entrare nella sua sessione critica; invece deve periodicamente controllare l'iglù, e perciò consuma tempo di elaborazione, rimanendo attivo (*busy*) mentre aspetta il proprio turno.

Dopo che un processo ha avuto accesso alla sua sezione critica e dopo aver completato l'azione, deve ritornare nell'iglù e scrivere sulla lavagna il numero dell'altro processo.

In termini formali, c'è una variabile globale condivisa:

```
var turno: 0 .. 1;
```

Il programma per i due processi è il seguente:

Processo 0

```
...
while turno ≠ 0 do { nulla };
< sezione critica >;
turno := 1;
...
```

Processo 1

```
...
while turno ≠ 1 do { nulla };
< sezione critica >;
turno := 0;
...
```

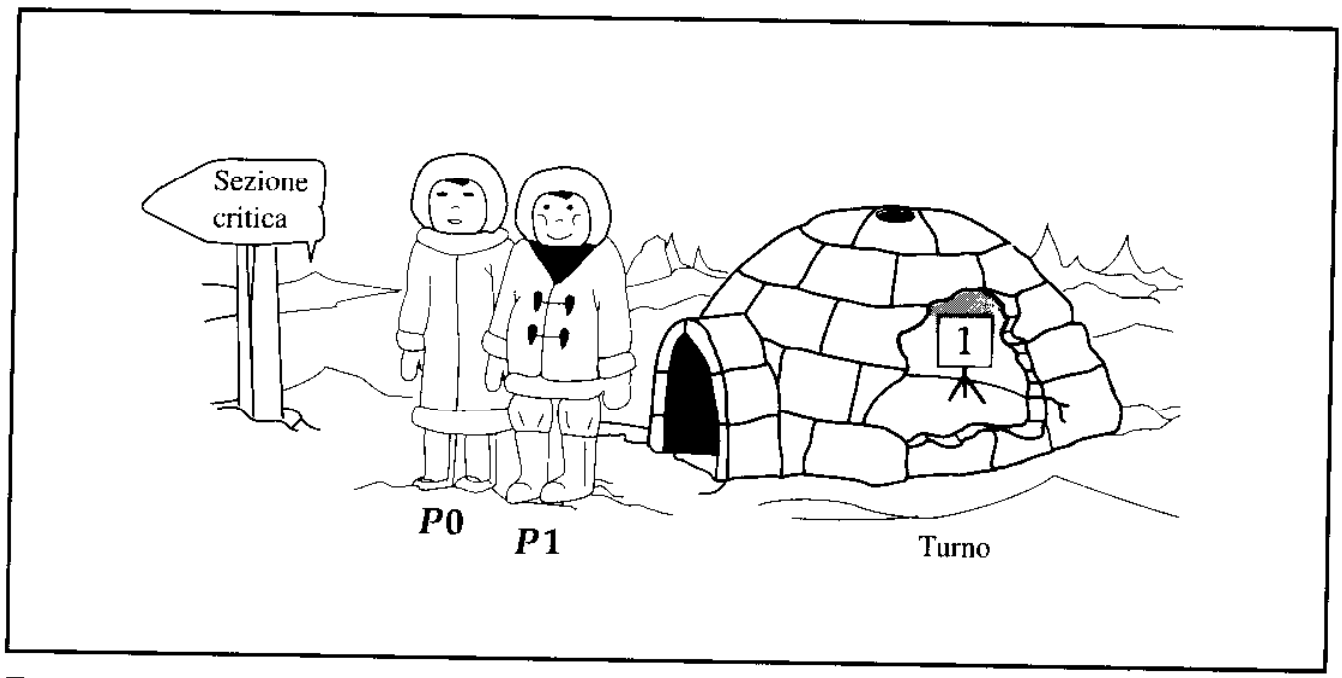


Figura 5.2 *Un iglù per la mutua esclusione*

Questa soluzione garantisce la proprietà di mutua esclusione ma ha due punti deboli. Il primo è che i processi devono osservare un'alternanza stretta per l'uso della sezione critica, così la velocità di esecuzione è data dal processo più lento. Se P0 usa la sua sezione critica solamente una volta all'ora ma P1 vorrebbe usare la propria 1000 volte all'ora, P1 è forzato a mantenere il ritmo di P0. Un problema molto più serio è che se un processo fallisce (ad esempio se è divorato da un orso polare mentre sta andando all'iglù), l'altro processo rimane bloccato per sempre: questo accade sia che il processo fallisca dentro alla sua sezione critica, sia fuori.

La costruzione usata in precedenza è quella di coroutine: le coroutine sono progettate per passarsi il controllo fra loro; sebbene rappresentino una tecnica di strutturazione utile nel caso di un solo processo, sono inadeguate per i processi concorrenti.

Secondo tentativo

Il problema del primo tentativo è che si scrive il nome del processo che può entrare nella sua sezione critica, quando in realtà è utile avere informazioni sullo stato di entrambi i processi. In effetti, ogni processo dovrebbe avere la propria chiave della sezione critica, così che, se un processo è eliminato da un orso polare, l'altro può ancora accedere alla propria sezione critica. Questa filosofia è illustrata nella Figura 5.3: ogni processo ora ha il suo iglù, e può guardare la lavagna dell'altro senza modificarla. Quando un processo desidera entrare nella propria sezione critica, controlla periodicamente la lavagna dell'altro finché non vede che c'è scritto "falso", il che indica che l'altro processo non è nella propria sezione critica; allora può dirigersi velocemente al proprio iglù e scrivere "vero" sulla lavagna, ed entrare nella sezione critica. Quando lascia la sezione critica deve scrivere "falso" sulla propria lavagna.

In questo caso la variabile globale condivisa è

```
var flag: array [0..1] of boolean;
```

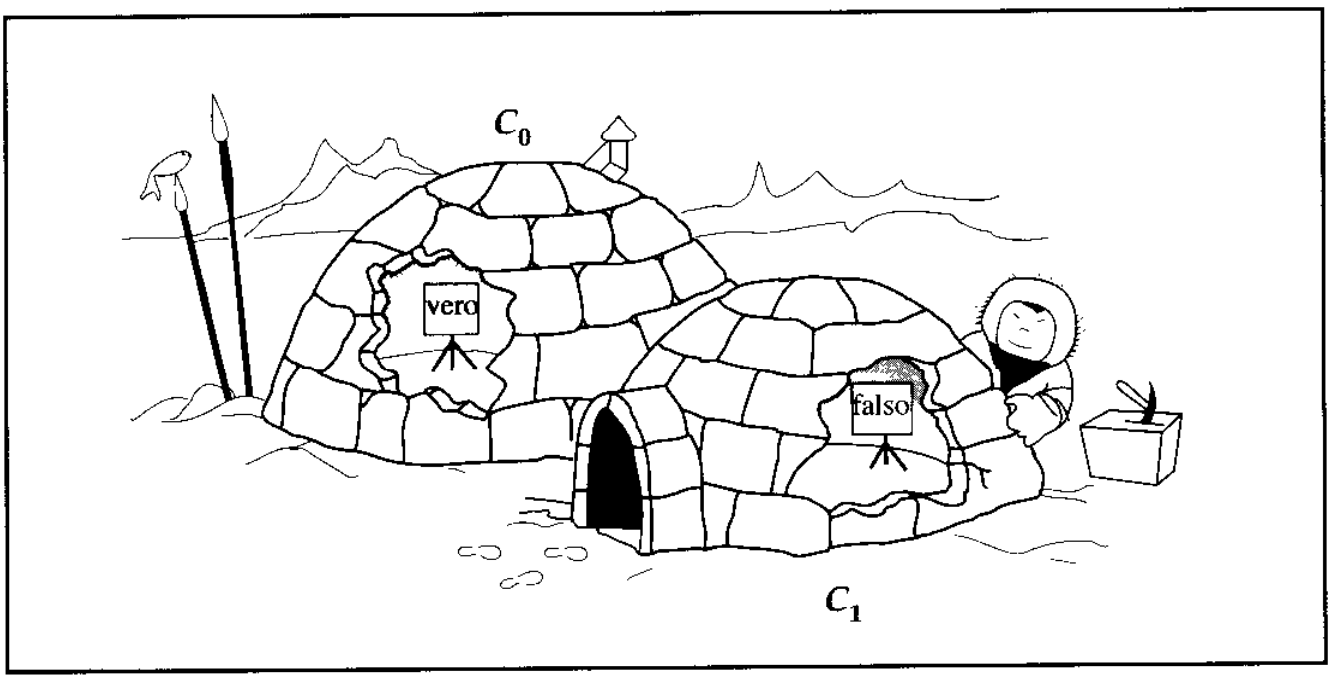


Figura 5.3 *Una soluzione per la mutua esclusione con due iglù*

ed è inizializzata a falso. Il programma per i due processi è il seguente

Processo 0

```
...
while flag[1] do { nulla };
flag[0] := true;
< sezione critica >;
flag[0] := false;
...
```

Processo 1

```
...
while flag[0] do { nulla };
flag[1] := true;
< sezione critica >;
flag[1] := false;
...
```

Se un processo fallisce all'esterno della sua sezione critica, compreso il codice per modificare il flag, allora l'altro processo non è bloccato: infatti, l'altro processo può entrare nella propria sezione critica tutte le volte che vuole, perché il flag dell'altro processo è sempre falso. Se invece un processo fallisce all'interno della propria sezione critica, o dopo aver messo il flag a vero prima di entrare, allora l'altro processo è bloccato per sempre.

Questa soluzione è al limite peggiore della precedente, perché non garantisce neppure la mutua esclusione. Si consideri questa sequenza:

P0 esegue il comando **while** e trova *flag[1]* falso.

P1 esegue il comando **while** e trova *flag[0]* falso.

P0 scrive vero in *flag[0]* e entra nella propria sezione critica.

P1 scrive vero in *flag[1]* e entra nella propria sezione critica.

Poiché entrambi i processi sono nella loro sezione critica, il programma non è corretto; il problema è che la soluzione proposta non è indipendente dalla velocità relativa di esecuzione dei processi.

Terzo tentativo

Il secondo tentativo è fallito perché un processo può cambiare il proprio stato dopo che l'altro processo lo ha letto ma prima che l'altro processo sia entrato nella sezione critica. Forse è possibile rimediare semplicemente scambiando due linee:

Processo 0

```
...
flag[0] := true;
while flag[1] do { nulla };
< sezione critica >;
flag[0] := false;
...
```

Processo 1

```
..
flag[1] := true;.
while flag[0] do { nulla };
< sezione critica >;
flag[1] := false;
...
```

Come prima, se un processo fallisce all'interno della propria sezione critica, compreso il codice per modificare il flag, allora l'altro processo è bloccato, e se un processo fallisce al-

l'esterno della propria sezione critica, l'altro processo non è bloccato.

Ora bisogna controllare che la mutua esclusione sia garantita, dal punto di vista di P0. Una volta che P0 ha messo *flag[0]* a vero, P1 non può entrare nella sua sezione critica finché P0 non è entrato e uscito dalla propria sezione critica. Quando P0 modifica il flag, è possibile che P1 sia già nella propria sezione critica, e in quel caso P0 sarà bloccato sul comando **while** finché P1 non esce dalla sezione critica. Lo stesso ragionamento si applica dal punto di vista di P1.

Questo garantisce la mutua esclusione ma crea un nuovo problema: se entrambi i processi mettono i propri flag a vero quando nessuno dei due ha ancora eseguito il comando **while**, allora ognuno penserà che l'altro sia entrato nella propria sezione critica, causando uno stallo.

Quarto tentativo

Nel terzo tentativo un processo modifica il proprio stato senza conoscere lo stato dell'altro processo. Si ha stallo perché ogni processo si ostina ad entrare nella propria sezione critica, e non esce da quella situazione. Si può provare a rimediare in modo che ogni processo sia più accomodante: ciascuno mette il flag a vero per indicare il desiderio di entrare nella propria sezione critica, ma è pronto a rimettere il flag a falso per lasciare spazio all'altro processo:

Processo 0

```
...
flag[0] := true;
while flag[1] do
  begin
       flag[0] := false;
       < breve pausa >;
       flag[0] := true;
  end;
< sezione critica >;
flag[0] := false;
...
```

Processo 1

```
...
flag[1] := true;
while flag[0] do
  begin
       flag[1] := false;
       < breve pausa >;
       flag[1] := true;
  end;
< sezione critica >;
flag[1] := false;
...
```

Siamo vicini ad una soluzione corretta, ma c'è ancora un problema: la mutua esclusione è ancora garantita, con un ragionamento simile a quello usato per il terzo tentativo, ma si consideri questa sequenza di eventi:

P0 scrive vero in *flag[0]*.
P1 scrive vero in *flag[1]*.
P0 controlla *flag[1]*.
P1 controlla *flag[0]*.
P0 scrive falso in *flag[0]*.
P1 scrive falso in *flag[1]*.
P0 scrive vero in *flag[0]*.
P1 scrive vero in *flag[1]*.

Se questa sequenza si ripetesse indefinitamente, nessun processo potrebbe entrare nella propria sezione critica. Volendo essere precisi, questo non è stallo, perché qualunque modifica della velocità relativa dei due processi romperebbe il ciclo e uno dei due riuscirebbe ad entrare nella sezione critica. Anche se è improbabile che un tale ciclo continui per molto tempo, è comunque una situazione possibile, così il quarto tentativo non può essere accettato.

Una soluzione corretta

Si è visto che bisogna osservare lo stato di entrambi i processi, tramite l'array *flag*, ma, come mostrato dal quarto tentativo, ciò non è sufficiente. Per evitare il problema di "mutua cortesia" che si è appena visto, è necessario imporre un ordine alle attività dei due processi. Per fare ciò si può usare la variabile *turno* del primo tentativo; in questo caso la variabile indica quale processo ha il diritto di insistere nel tentativo di entrare nella propria sezione critica.

Questa soluzione può essere descritta in termini di iglù facendo riferimento alla Figura 5.4. Ora c'è un iglù "arbitro" con una lavagna chiamata "turno". Quando P0 vuole entrare nella propria sezione critica, mette il proprio flag a vero, e poi va a controllare il flag di P1: se è falso, P0 può entrare nella sezione critica. Altrimenti P0 va a consultare l'arbitro: se trova che *turno* = *0* sa che è il suo turno di insistere, e controlla periodicamente l'iglù di P1. P1 ad un certo punto noterà che è il suo turno di rinunciare, e scriverà falso sulla propria lavagna, permettendo a P0 di entrare. Dopo aver usato la propria sezione critica, P0 mette il proprio flag a falso per liberare la sezione critica, e mette *turno* a 1 per trasferire a P1 il diritto di insistere.

La Figura 5.5 mostra l'algoritmo di Dekker. Il costrutto **parbegin** P1; P2; ...; Pn **parend** ha l'effetto di sospendere l'esecuzione del programma principale ed iniziare l'esecuzione concor-

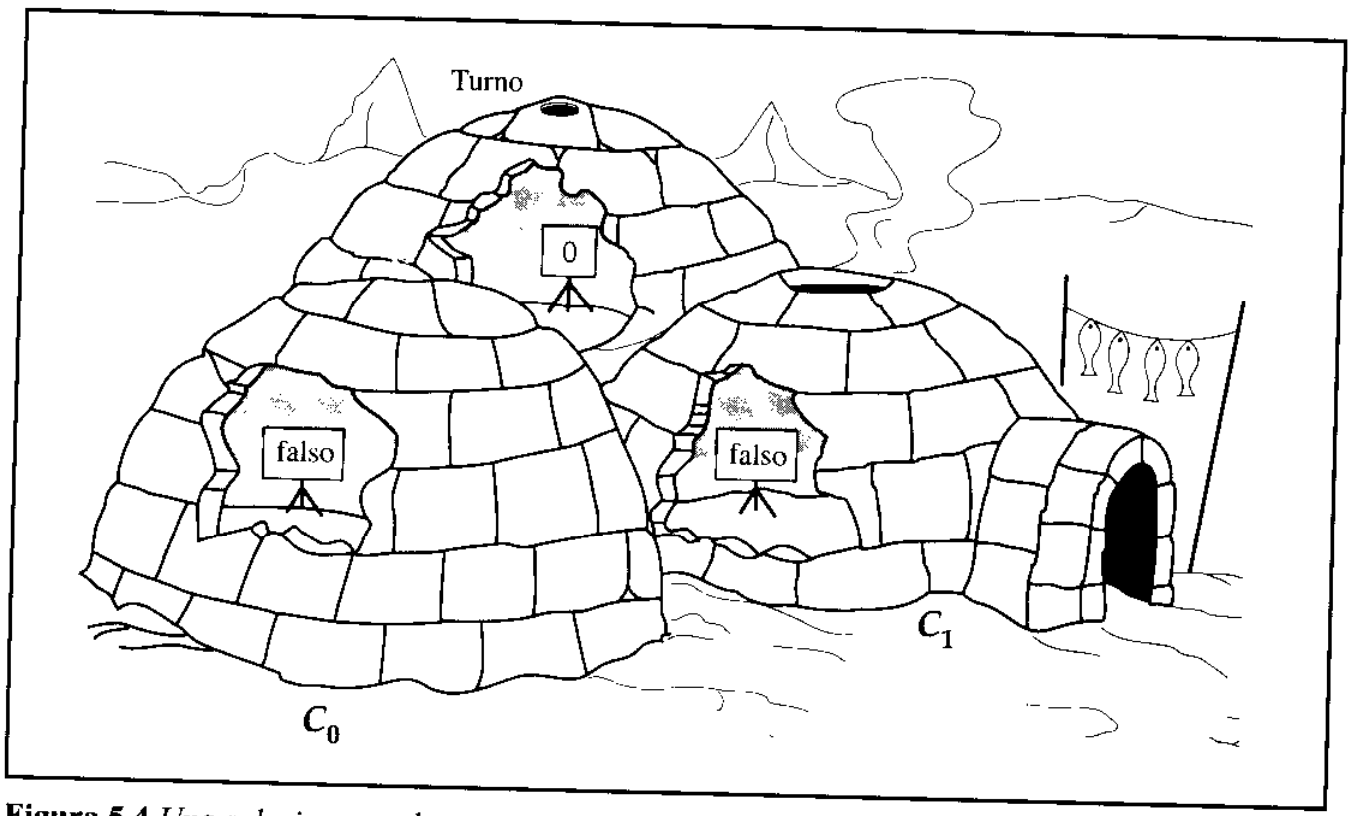


Figura 5.4 *Una soluzione per la mutua esclusione con tre iglù*

```
var flag: array [0..1] of boolean;
   turno: 0 .. 1;
procedure P0;
begin
  repeat
     flag [0] := true;
     while flag [1] do if turno = 1 then
        begin
           flag [0] := false;
           while turno = 1 do { nulla };
            flag [0] := true;
        end;
     < sezione critica >;
     turno := 1;
     flag[0] := false;
     < resto del programma >
  forever
end;
procedure P1;
begin
  repeat
     flag [1] := true;
     while flag[0] do if turno = 0 then
        begin
           flag [1] := false;
           while turno = 0 do [ nulla };
           flag [1] := true;
        end;
     < sezione critica >;
     turno := 0;
     flag [1] := false;
     < resto del programma >
  forever
end;
begin
  flag [0] := false;
  flag [1] := false;
  turno := 1;
  parbegin
     P0; P1
  parend
end.
```

Figura 5.5 *L'algoritmo di Dekker*

rente delle procedure P1, P2, ..., Pn. Quando tutti i processi hanno terminato, viene ripristinato il programma principale. La dimostrazione di correttezza dell'algoritmo di Dekker è lasciata per esercizio (si veda il Problema 5.6).

L'algoritmo di Peterson

L'algoritmo di Dekker risolve il problema della mutua esclusione con un programma piuttosto complesso: è difficile da seguire e la dimostrazione della correttezza è complessa. Peterson [PETE81] fornisce una soluzione semplice ed elegante: come prima l'array globale *flag* indica la posizione dei vari processi rispetto alla mutua esclusione, e la variabile globale *turno* risolve i conflitti. L'algoritmo è nella Figura 5.6.

Si vede facilmente che la mutua esclusione è garantita. Si consideri il processo P0: quando ha messo *flag*[0] a vero, P1 non può entrare nella propria sezione critica; se P1 è già nella sezione critica, allora *flag*[1] è vero e P0 non può entrare nella propria sezione critica. Inoltre non è possibile che i due processi si blocchino a vicenda: se P0 è bloccato nel suo ciclo **while**, allora *flag*[1] è vero e *turno* = 1; P0 può entrare nella propria sezione critica quando *flag*[1] diventa falso, oppure *turno* diventa 0. Si considerino questi tre casi esaustivi:

```
var flag: array [0..1] of boolean;
    turno: 0 .. 1;
procedure P0;
begin
   repeat
      flag [0] := true;
      turno := 1;
      while flag [1] and turno = 1 do { nulla };
      < sezione critica >;
      flag[0] := false;
      < resto del programma >
   forever
end;
procedure P1;
begin
   repeat
      flag [1] := true;
      turno := 0;
      while flag [0] and turno = 0 do { nulla };
      < sezione critica >;
      flag[1] := false;
      < resto del programma >
   forever
end;
begin
   flag [0] := false;
   flag [1] := false;
   turno := 1;
   parbegin
      P0; P1
   parend
end.
```

Figura 5.6 *L'algoritmo di Peterson per due processi*

1. P1 non vuole entrare nella propria sezione critica: questo è impossibile perché ciò implica che *flag*[1] è falso.
2. P1 sta aspettando di entrare nella sezione critica: anche questo caso è impossibile perché se *turno* = 1 allora P1 può entrare.
3. P1 sta usando ripetutamente la propria sezione critica e quindi sta monopolizzando l'accesso: anche questo non può accadere, perché P1 è obbligato a dare a P0 un'opportunità, mettendo *turno* a 0 prima di ogni tentativo di accesso alla sezione critica.

Quindi c'è una soluzione semplice al problema della mutua esclusione per due processi; inoltre l'algoritmo di Peterson può essere facilmente generalizzato al caso di *n* processi[HOFR80].

5.3 Mutua esclusione: supporto hardware

Disabilitare le interruzioni

In una macchina a singolo processore, i processi concorrenti non possono sovrapporsi: possono solamente alternarsi; pertanto un processo continua l'esecuzione fino a quando non chiama un servizio del sistema operativo o viene interrotto. Quindi per garantire la mutua esclusione è sufficiente evitare che un processo venga interrotto: questo può essere realizzato con apposite primitive del kernel di sistema per abilitare e disabilitare gli interrupt. In definitiva, un processo può garantire la mutua esclusione in questo modo (vedi Figura 5.1):

```
repeat
   < disattiva le interruzioni > ;
   < sezione critica > ;
   < attiva le interruzioni > ;
   < resto del programma >
forever.
```

Poiché la sezione critica non può essere interrotta, la mutua esclusione è garantita, comunque questo approccio ha un prezzo molto alto: l'efficienza può peggiorare perché il processore non può alternare i programmi liberamente. Un altro problema è che tale approccio non funziona in un'architettura multiprocessore: quando il sistema è composto da vari processori, due o più processi possono essere eseguiti contemporaneamente, e in tal caso disattivare gli interrupt non garantisce la mutua esclusione.

Istruzioni macchina speciali

In una configurazione multiprocessore, molti processori possono condividere l'accesso alla memoria centrale: in questo caso non c'è una relazione gerarchica, ma i processori funzionano indipendentemente e c'è una relazione di parità fra di essi. Non è possibile garantire la mutua esclusione con un meccanismo basato sugli interrupt.

A livello hardware, come visto, l'accesso ad una locazione di memoria esclude qualunque altro accesso alla stessa locazione; basandosi su questo fatto, i progettisti hanno proposto varie istruzioni macchina che effettuano due azioni in modo atomico, come lettura e scrittura, o lettura e test, di una locazione di memoria in un solo ciclo di istruzione. Poiché queste azioni vengono effettuate in un solo ciclo, non sono soggette ad interferenze con altre istruzioni.

In questa sezione si analizzeranno le due istruzioni più comuni, e si rimanda a [RAYN86] e [STON93] per altri esempi.

L'istruzione test-and-set

Si può definire l'istruzione test-and-set in questo modo:

```
function testset (var i: integer): boolean;
begin
  if  i = 0 then
      begin
          i := 1;
          testset := true;
      end
  else  testset := false
end
```

L'istruzione legge il valore dell'argomento *i*: se vale 0, lo rimpiazza con 1 e ritorna vero, altrimenti lascia il valore inalterato e ritorna falso. L'intera funzione viene eseguita atomicamente: cioè non è soggetta ad interruzioni.

La Figura 5.7a mostra un protocollo per la mutua esclusione basato sull'uso di questa istruzione. La variabile condivisa *serratura* viene inizializzata a 0. L'unico processo che può entrare nella propria sezione critica è quello che trova *serratura* uguale a 0. Tutti gli altri processi che cercano di entrare nella propria sezione critica entrano in stato di attesa attiva. Quando un processo lascia la sezione critica, mette *serratura* a 0, e a quel punto uno ed un solo processo in attesa potrà entrare nella propria sezione critica. La scelta del processo dipende da quello che eseguirà l'istruzione test-and-set per primo.

L'istruzione di scambio

L'istruzione di scambio si può definire in questo modo:

```
procedure scambio (var r: registro; var m: memoria);
var temp;
begin
  temp := m;
  m := r;
  r := temp;
end.
```

```
program  mutuaesclusione;
const  n =  ... ; (* numero di processi *);
var  serratura:  integer;
procedure P(i: integer);
begin
   repeat
      repeat { nulla } until testset(serratura);
       < sezione critica >;
      serratura := 0;
      < resto del programma >
   forever
 end;
begin (*  programma principale *)
   serratura := 0;
   parbegin
     P(1);
     P(2);
     ...
     P(n)
   parend
end.
```

(a) Istruzione Test-And-Set

```
program  mutuaesclusione;
const  n =  ... ; (* numero di processi *);
var  serratura:  integer;
procedure P(i: integer);
var  chiavei:  integer;
begin
   repeat
     chiavei := 1;
       repeat scambio(chiavei, serratura) until chiavei = 0;
        < sezione critica >;
       scambio(chiavei, serratura)
       < resto del programma >
   forever
 end;
begin (*  programma principale *)
   serratura := 0;
   parbegin
     P(1);
     P(2);
     ...
     P(n)
   parend
end.
```

(b) Istruzione di Scambio

Figura 5.7 *Supporto hardware alla mutua esclusione*

L'istruzione scambia il contenuto di un registro con quello di una locazione di memoria; durante l'esecuzione l'accesso alla locazione di memoria viene impedito a tutte le altre istruzioni.

La Figura 5.7b mostra un protocollo di mutua esclusione basato sull'uso di questa istruzione: una variabile condivisa *serratura* viene inizializzata a 0 e ogni processo usa una variabile locale *chiave* che viene inizializzata a 1; l'unico processo che può entrare nella propria sezione critica è quello che trova *serratura* uguale a 0, e impedisce agli altri l'accesso alla loro sezione critica mettendo *serratura* a 1. Quando un processo lascia la sezione critica, rimette *serratura* a 0, così gli altri processi possono accedere alla sezione critica.

Si noti che l'equazione che segue è sempre vera per il modo in cui le variabili vengono inizializzate e per la natura dell'algoritmo di scambio:

$$serratura + \sum_i chiave_i = n$$

Se *serratura* = 0 nessun processo è nella sezione critica, e se *serratura* = 1 allora esattamente un processo è nella sezione critica: quello la cui variabile *chiave* vale, 0.

Proprietà dell'approccio con istruzioni speciali

L'uso di un'istruzione macchina speciale per garantire la mutua esclusione ha vari vantaggi:

- Si può applicare a qualunque numero di processi, su un singolo processore o su un multiprocessore con memoria condivisa.
- È semplice e quindi facile da verificare.
- Si può usare per fornire più di una sezione critica: ciascuna sezione avrà una propria variabile.

Ci sono anche degli svantaggi importanti:

- Bisogna usare la tecnica dell'attesa attiva, quindi mentre un processo aspetta di avere accesso alla sezione critica, usa il tempo di esecuzione del processore.
- È possibile che si verifichi starvation: quando un processo abbandona la sezione critica e ci sono vari processi in attesa, la scelta del processo da attivare è arbitraria, quindi è possibile che un processo debba aspettare per un tempo illimitato.
- È possibile che si verifichi uno stallo; si consideri il caso di un singolo processore, in cui un processo P1 esegue l'istruzione speciale (test-and-set o scambio) ed entra nella sezione critica; dopo di che viene interrotto per concedere il processore a P2, che ha una priorità più alta. Se P2 tenta di accedere alla stessa risorsa di P1, riceverà un rifiuto a causa del meccanismo di mutua esclusione, così entrerà in un ciclo di attesa attiva. Comunque P1 non verrà mai riattivato perché la sua priorità è più bassa di quella di P2, che è attivo.

A causa delle limitazioni viste sia nell'approccio software sia in quello hardware, è necessario cercare altri meccanismi.

5.4 Semafori

Nel seguito verranno studiati i meccanismi dei sistema operativo e dei linguaggi di programmazione per gestire la concorrenza. In questa sezione vengono introdotti i semafori, e nelle due successive i monitor e lo scambio di messaggi.

Il primo contributo significativo allo studio dei problemi dei processi concorrenti è il lavoro di Dijkstra nel 1965 [DIJK65]. Dijkstra era interessato alla progettazione di un sistema operativo come un insieme di processi sequenziali che cooperano, e allo sviluppo di meccanismi efficienti e affidabili per il supporto alla cooperazione; una volta implementati dal processore o dal sistema operativo, essi possono essere usati dai processi dell'utente, in modo semplice.

Il principio fondamentale è il seguente: due o più processi possono cooperare attraverso semplici segnali, in modo che un processo si fermi ad una locazione ben precisa finché non riceve un segnale specifico. Qualunque esigenza complessa di coordinazione può essere soddisfatta con un uso appropriato dei segnali. Tali segnalazioni sono fatte da un tipo speciale di variabile: il semaforo. Per trasmettere un segnale tramite il semaforo *s*, un processo esegue la primitiva *signal(s)*, e per ricevere il segnale utilizza *wait(s)*; se il segnale corrispondente non è stato ancora trasmesso, il processo viene sospeso finché non avviene la trasmissione[2].

Per raggiungere l'effetto desiderato, un semaforo si può vedere come una variabile che ha un valore intero e tre operazioni associate:

1. Inizializzazione con un valore non negativo.
2. L'operazione *wait* decrementa il valore del semaforo; se il valore diventa negativo, il processo che esegue la *wait* viene bloccato.
3. L'operazione *signal* incrementa il valore del semaforo; se il valore non è positivo allora uno dei processi bloccati sull'operazione *wait* viene liberato.

A parte queste tre operazioni, non c'è altro modo per leggere lo stato o modificare i semafori.

La Figura 5.8 suggerisce una definizione più formale delle primitive per i semafori; si suppone che *wait* e *signal* siano atomiche, cioè che non possano essere interrotte e che ogni routine sia trattata come un passo non scomponibile. Una versione particolare, il semaforo binario, si trova nella Figura 5.9. Un semaforo binario può assumere solo i valori 0 e 1; l'implementazione dovrebbe essere più semplice e si può dimostrare che il potere espressivo è uguale a quello del semaforo generico (si veda il Problema 5.13).

Per implementare i semafori (binari o generici) si usa una coda che contiene i processi in attesa sul semaforo. La definizione non stabilisce l'ordine in cui i processi vengono eliminati dalla coda, ma la politica più equa è quella detta first-in-first-out (*primo arrivato, primo servito*): il processo che ha aspettato più a lungo è quello che viene liberato per primo dalla coda, e l'unico vincolo è che nessun processo deve essere lasciato nella coda di un semaforo indefinitamente per dare la precedenza ad altri processi.

---

[2] Nel lavoro originale di Dijkstra, e spesso in letteratura, la lettera P è usata per wait, e V per signal, dalle iniziali delle parole Olandesi *proberen* (provare) e *verhogen* (incrementare).

```
type semaforo = record
                  contatore:  integer;
                  coda: list of process
                end;
var s: semaforo;

wait(s):
s.contatore := s.contatore - 1
if s.contatore < 0
   then begin
      metti questo processo in s.coda;
      blocca questo processo
    end;

signal(s):
s.contatore := s.contatore + 1;
 if s.contatore ≤ 0 then begin
   togli un processo P da s.coda;
   metti P in stato Ready
end;
```

Figura 5.8 *Definizione delle primitive per i semafori*

```
type semaforo_binario = record
                          valore:  (0,1);
                          coda: list of process
                        end;
var s: semaforo_binario;

waitB(s):
if s.valore = 1
   then
     s.valore = 0;
   else begin
      metti questo processo in s.coda;
      blocca questo processo
    end;

signalB(s):
 if s.coda è vuota then
     s.valore = 1;
   else begin
     togli un processo P da s.coda;
     metti P in stato Ready
end;
```

Figura 5.9 *Definizione delle primitive per i semafori binari*

Mutua esclusione

La Figura 5.10 mostra una soluzione semplice del problema della mutua esclusione usando un semaforo *s* (vedi Figura 5.1). Si considerino *n* processi, dove l'*i*-esimo processo è indicato come *P(i)*. Ogni processo esegue *wait(s)* subito prima della sezione critica; se il valore di *s* diventa negativo, il processo viene sospeso. Se il valore è 1, viene messo a 0 e il processo entra immediatamente nella sezione critica; poiché *s* non ha più un valore positivo, nessun altro processo potrà entrare nella propria sezione critica.

Il semaforo viene inizializzato a 1, così il primo processo che esegue *wait* potrà entrare nella sezione critica e il valore verrà messo a 0; in seguito, se un altro processo cercherà di entrare nella sezione critica, la troverà occupata, verrà bloccato, e il valore di *s* diventerà –1. Non c'è limite al numero di processi che possono tentare di entrare nella propria sezione critica, e ogni tentativo diminuirà di 1 il valore di *s*. Quando il processo iniziale esce dalla sezione critica, *s* viene incrementato e uno dei processi in attesa (se ce ne sono) verrà tolto dalla coda dei processi bloccati associata al semaforo, e messo in stato Ready: in questo modo, potrà entrare nella sezione critica non appena sarà attivato dal sistema operativo.

L'algoritmo di mutua esclusione con semafori si può illustrare con il modello dell'iglù (Figura 5.11): oltre alla lavagna, l'iglù contiene un congelatore. Un processo entra nell'iglù per effettuare una *wait*; decrementa di 1 il valore scritto sulla lavagna e se il valore non è negativo può entrare nella sezione critica, altrimenti entra nel congelatore per ibernarsi: ciò libera spazio nell'iglù, così un altro processo può entrare. Quando un processo ha completato la sua sezione critica, entra nell'iglù ed effettua una *signal*, incrementando di 1 il valore della lavagna; se il risultato non è positivo, toglie un processo dal congelatore.

```
program  mutuaesclusione;
const  n =  ... ; (* numero di processi *);
var  s:  semaforo (:= 1);
procedure P(i: integer);
begin
   repeat
     wait(s);
        < sezione critica >;
       signal(s);
       < resto del programma >
   forever
 end;
begin (*  programma principale *)
   parbegin
     P(1);
     P(2);
     ...
     P(n)
   parend
end.
```

Figura 5.10 *Mutua esclusione con semafori*

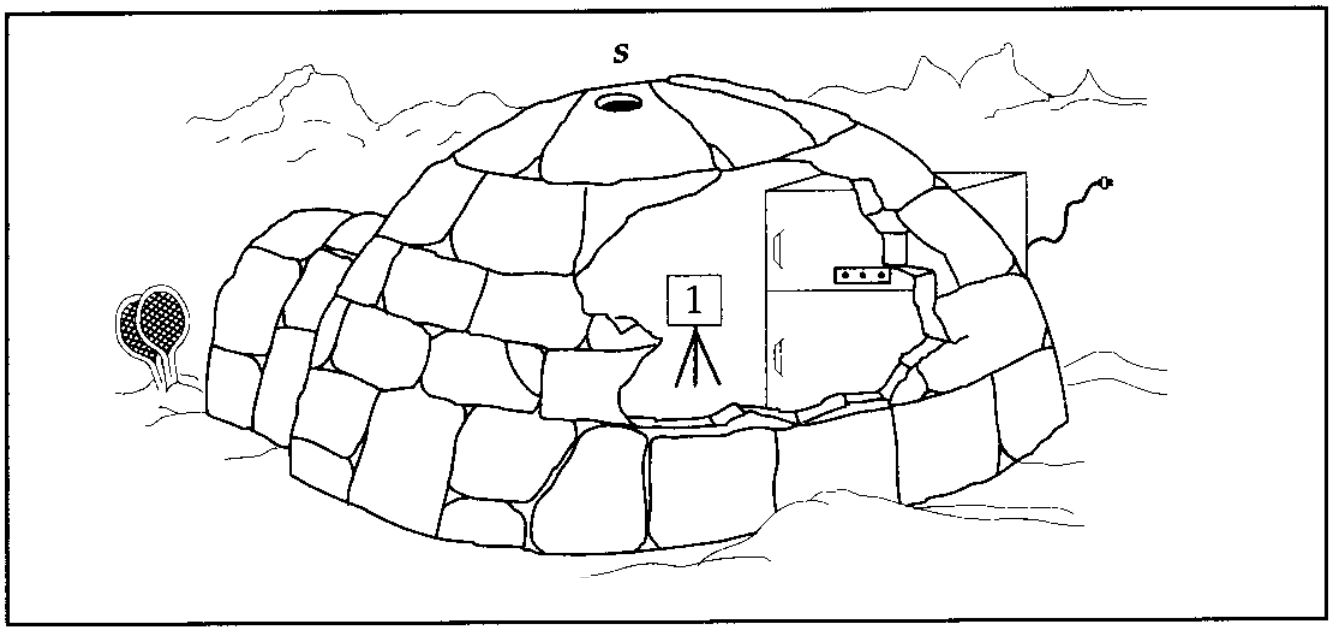


Figura 5.11 *Un iglù semaforo*

Il programma in Figura 5.10 può gestire in modo analogo il caso in cui si permette che più di 1 processo per volta sia all'interno della propria sezione critica: è sufficiente inizializzare il semaforo con il valore desiderato; dopodiché, in ogni istante il valore di *s.contatore* si può interpretare così:

- *s. contatore* ≥ 0 significa che *s. contatore* è il numero di processi che possono eseguire una *wait(s)* senza essere bloccati (se nessuna *signal* viene eseguita nel frattempo).
- *s. contatore* < 0 significa che il valore assoluto di *s. contatore* è il numero di processi sospesi nella coda *s.coda*.

Il problema del produttore/consumatore

In questa sezione si studia il problema più comune dei processi concorrenti: il problema del produttore/consumatore. La formulazione generale è la seguente: ci sono uno o più produttori che generano qualche tipo di dato (record, caratteri) e lo mettono in un buffer, e c'è un solo consumatore che prende gli elementi del buffer uno alla volta. Il sistema deve impedire la sovrapposizione di operazioni sul buffer, cioè un solo agente (il produttore o il consumatore) alla volta può accedere al buffer. Si prenderanno in considerazione varie soluzioni al problema per mostrare la potenza e i punti deboli dei semafori.

Per cominciare, si supponga che il buffer sia infinito e si componga di un array di elementi. In astratto, si può definire una funzione per il produttore in questo modo:

```
produttore:
   repeat
        produci v;
        b[in] := v;
        in := in + 1
   forever;
```

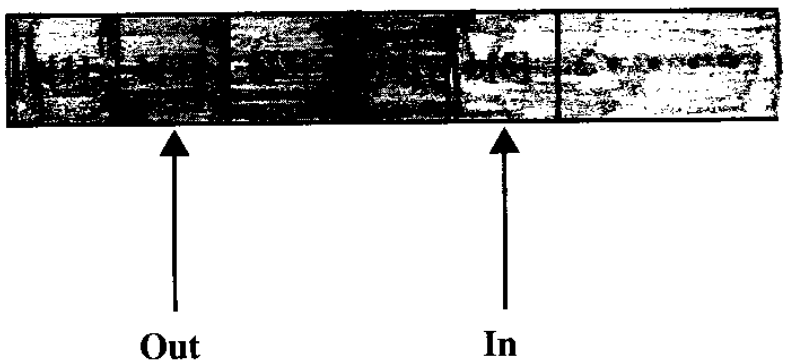
Analogamente per il consumatore:

```
consumatore:
  repeat
      while in ≤ out do { nulla };
      w := b[out];
      out := out + 1;
      consuma w;
  forever;
```

La Figura 5.12 mostra la struttura del buffer *b*. Il produttore può generare elementi e aggiungerli al buffer senza vincoli di velocità; ad ogni aggiunta è incrementato un indice (*in*) nel buffer. Il consumatore ha un comportamento simile, ma deve prestare attenzione a non tentare di leggere il buffer quando è vuoto, quindi prima di procedere deve assicurarsi che il consumatore si trovi più avanti di lui ($in > out$).

Un primo tentativo di implementazione con semafori binari si trova nella Figura 5.13. Invece di gestire gli indici *in* e *out*, si può semplicemente tenere traccia del numero di elementi contenuti nel buffer usando la variabile n ($= in - out$). Si usano due semafori: *s* per garantire la mutua esclusione, e *ritardo* per forzare il consumatore ad aspettare se il buffer è pieno.

Questa soluzione è piuttosto semplice: il produttore è libero di aggiungere elementi al buffer in qualsiasi momento. Effettua *waitB(s)* prima di aggiungere un elemento e *signalB(s)* dopo, per impedire un eventuale accesso del consumatore o di un altro produttore durante l'operazione di aggiunta. Inoltre, nella sezione critica, il produttore incrementa il valore di n; se $n = 1$ allora il buffer era vuoto prima dell'aggiunta, così il produttore effettua *signalB(ritardo)* per avvisare il consumatore. Il consumatore inizia facendo *waitB(ritardo)* in modo da aspettare che il primo elemento sia prodotto; in seguito prende un elemento e decrementa n all'interno della sezione critica. Se il produttore è più veloce del consumatore (una situazione comune), quest'ultima si bloccherà raramente sul semaforo *ritardo*, perché solitamente n sarà positivo. Quindi il produt-



Nota: le aree grigie indicano le porzioni del buffer che sono occupate

Figura 5.12 *Buffer infinito per il problema del produttore/consumatore*

```
program produttore_consumatore;
var    n: integer;
       s: semaforo (:= 1); (* binario *)
       ritardo: semaforo (:= 0); (* binario*)
procedure produttore;
begin
   repeat
      produci;
      waitB(s);
      aggiungi;
      n := n + 1;
      if n=1 then signalB(ritardo);
      signalB(s)
   forever
end;
 procedure consumatore;
begin
   waitB(ritardo);
   repeat
      waitB(s);
      prendi un elemento;
      n := n - 1;
      signalB(s);
      consuma;
      if n=0 then waitB(ritardo)
   forever
end;
begin (* programma principale*)
    n := 0;
   parbegin
      produttore; consumatore
   parend
end.
```

Figura 5.13 *Soluzione non corretta del problema del produttore/consumatore con semafori binari e buffer infinito*

tore e il consumatore potranno lavorare senza problemi.

Comunque c'è un difetto in questo programma: quando il consumatore ha svuotato il buffer, deve aspettare che il produttore aggiunga un elemento al buffer. Questo è lo scopo del comando **if** $n = 0$ **then** *waitB(ritardo)*. Si consideri allora la sequenza della Tabella 5.2: alla riga 6 il consumatore non esegue la *waitB*; in realtà il buffer è stato svuotato e n è stato messo a 0 (riga 4), ma il produttore ha incrementato n prima che il consumatore potesse leggerne il valore alla riga 6. Il risultato è una *signalB* che non segue nessuna *waitB*: il valore -1 che viene assegnato ad n (riga 9) significa che il consumatore ha preso un elemento inesistente dal buffer vuoto. Spostare il controllo su $n = 0$ all'interno della sezione critica del consumatore non sarebbe una soluzione, perché potrebbe causare uno stallo (ad esempio alla riga 3).

Tabella 5.2 *Possibile sequenza di esecuzione del programma della Figura 5.13*

	Azione	*n*	Ritardo
1	Inizialmente	0	0
2	Produttore: sezione critica	1	1
3	Consumatore: waitB (ritardo)	1	0
4	Consumatore: sezione critica	0	0
5	Produttore: sezione critica	1	1
6	Consumatore: if n = 0 then waitB(ritardo)	1	1
7	Consumatore: sezione critica	0	1
8	Consumatore: if n = 0 then waitB(ritardo)	0	0
9	Consumatore: sezione critica	−1	0

Una soluzione al problema, mostrata nella Figura 5.14, è l'introduzione di una variabile ausiliaria che viene assegnata all'interno della sezione critica del consumatore, ed è usata dopo. Un'analisi accurata della logica del programma dovrebbe convincere che non ci può essere stallo.

Utilizzando i semafori generici (detti anche semafori a contatore) si ottiene una soluzione più pulita, mostrata nella Figura 5.15. La variabile *n* ora è un semaforo, e ancora il suo valore è uguale al numero di elementi nel buffer. Si supponga che nel trascrivere il programma si scambino per errore le operazioni *signal(s)* e *signal(n)*: ciò richiederebbe l'esecuzione delle *signal(n)* nella sezione critica del produttore, senza interruzioni da parte del consumatore o di altri produttori. Cambierebbe l'effetto del programma? No, perché il consumatore in ogni caso dovrebbe aspettare su entrambi i semafori prima di procedere.

Si supponga ora che le operazioni *wait(n)* e *wait(s)* vengano scambiate per errore: ciò produce un errore fatale, infatti, se il consumatore entra nella sezione critica quando il buffer è vuoto (*n.contatore = 0*), nessun produttore può aggiungere elementi al buffer, quindi si ha stallo. Questo è un buon esempio degli aspetti subdoli dei semafori e delle difficoltà di realizzazione di un progetto corretto.

Per finire, aggiungiamo una restrizione realistica al problema del produttore/consumatore: il buffer ha dimensione finita. Il buffer è visto come un sistema di memorizzazione circolare (Figura 5.16), e i valori dei puntatori sono espressi modulo la dimensione del buffer. Le funzioni del produttore e del consumatore si possono esprimere come segue (le variabili *in* e *out* vengono inizializzate a 0):

```
produttore:
repeat
   produci v;
   while ( (in + 1) mod n = out) do {nulla};
   b[in] := v;
   in := (in + 1) mod n;
forever;
```

```
program produttore_consumatore;
var    n: integer;
       s: semaforo (:= 1); (* binario*)
       ritardo: semaforo (:= 0); (* binario*)
procedure produttore;
begin
   repeat
      produci;
      waitB(s);
      aggiungi;
      n := n + 1;
      if n=1 then signalB(ritardo);
      signalB(s)
   forever
end;
procedure consumatore;
var m : integer;  (* variabile locale *)
begin
   waitB(ritardo);
   repeat
      waitB(s);
      prendi un elemento;
      n := n - 1;
      m := n;
      signalB(s);
      consuma;
      if m=0 then waitB(ritardo)
   forever
end;
begin (* programma principale*)
    n := 0;
   parbegin
      produttore; consumatore
   parend
end.
```

Figura 5.14 *Soluzione corretta del problema del produttore/consumatore con buffer infinito utilizzando semafori binari*

```
consumatore:
repeat
  while in = out do {nulla};
  w := b[out];
  out := (out + 1) mod n;
  consuma w;
forever;
```

```
program produttore_consumatore;
var     n: semaforo (:= 0);
        s: semaforo (:= 1);
procedure produttore;
begin
    repeat
        produci;
        wait(s);
        aggiungi;
        signal(s);
        signal(n)
    forever
  end
  procedure consumatore;
begin
    repeat
        wait(n);
        wait(s);
        prendi un elemento;
        signal(s);
        consuma
    forever
end;
begin (* programma principale *)
    parbegin
        produttore; consumatore
     parend
end.
```

Figura 5.15 *Soluzione del problema del produttore/consumatore con buffer infinito utilizzando semafori generici*

La Figura 5.17 mostra una soluzione con semafori generici; il semaforo *e* è stato aggiunto per tenere traccia del numero di posizioni libere nel buffer.

Implementazione dei semafori

Come detto prima, è necessario che le operazioni *wait* e *signal* vengano implementate come primitive atomiche. Una soluzione ovvia è implementarle in hardware o firmware; in alternativa sono stati proposti diversi schemi. L'essenza del problema è la mutua esclusione: un solo processo alla volta può accedere al semaforo con un'operazione di *wait* o *signal*, pertanto è possibile usare uno schema software qualunque, come l'algoritmo di Dekker o quello di Peterson, ma a spesa di un sovraccarico di elaborazione. Un'alternativa è di usare uno schema hardware per la mutua esclusione, come ad esempio un'istruzione test-and-set, mostrata in Figura 5.18a. In quest'implementazione, il semaforo è ancora di tipo **record** come in Figura 5.8, ma contiene una componente intera: *s.flag*. Bisogna ammettere che questo crea una forma di attesa attiva,

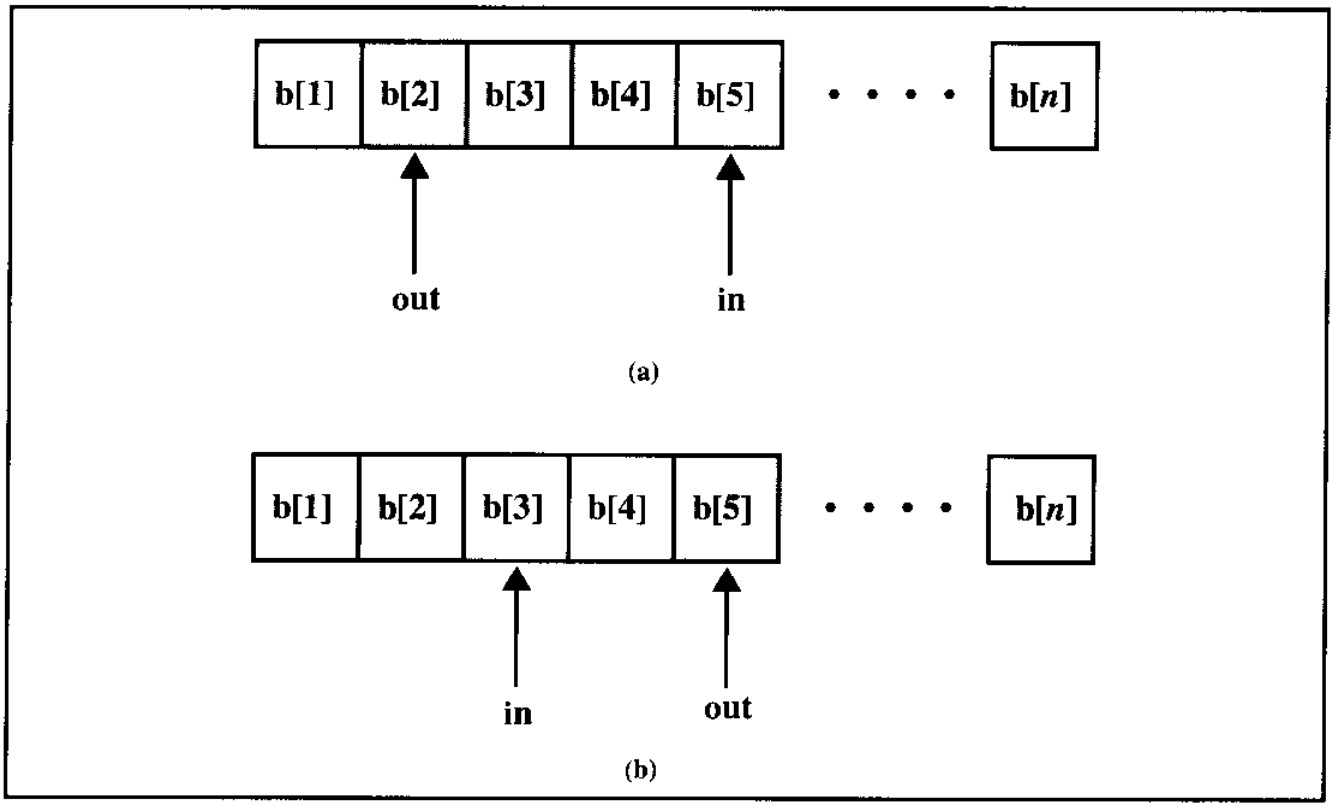


Figura 5.16 *Buffer circolare per il problema del produttore/consumatore*

comunque le operazioni di *wait* e *signal* sono relativamente brevi, quindi il tempo di attesa attiva non è elevato.

Per un sistema a singolo processore si possono semplicemente disabilitare gli interrupt durante un'operazione di *wait* o *signal*, come suggerito nella Figura 5.18b; ancora una volta, vista la durata relativamente breve di queste operazioni, si può affermare che l'approccio è ragionevole.

Il problema del barbiere

Un altro esempio dell'uso dei semafori per implementare la concorrenza è il problema del barbiere[3]. Quest'esempio è istruttivo, perché i problemi incontrati per gestire le risorse del negozio di barbiere sono simili a quelli di un sistema operativo reale.

Nel negozio del barbiere ci sono tre sedie, tre barbieri, una sala d'aspetto con un divano per quattro persone e spazio per altre persone in piedi (Figura 5.19); le norme antincendio limitano il numero massimo di persone presenti contemporaneamente a 20. Si supponga ad esempio che il negozio debba soddisfare 50 clienti.

Quando il negozio contiene il numero massimo di clienti, nessun nuovo cliente può entrare. Una volta entrato, il cliente si siede sul divano o rimane in piedi se non c'è posto; quando si libera un barbiere, il primo ad essere servito è il cliente che è rimasto più a lungo sul divano, e se ci sono dei clienti in piedi, quello che è rimasto in piedi più a lungo si siede sul divano. Quando un barbiere ha finito con un cliente, deve farsi pagare, ma c'è un solo registratore di cassa, così

[3] Sono grato al Professor Ralph Hilzer della California State University per aver fornito questo problema.

```
program bufferlimitato;
const  dimensione_del_buffer = . . .;
var    s: semaforo (:= 1);
       n: semaforo (:= 0);
       e: semaforo (:= dimensione_del_buffer);
procedure produttore;
begin
   repeat
      produci;
      wait(e);
      wait(s);
      aggiungi;
      signal(s);
      signal(n)
   forever
end;
procedure consumatore;
begin
   repeat
      wait(n);
      wait(s);
      prendi un elemento;
      signal(s);
      signal(e);
      consuma
   forever
end;
begin (* programma principale *)
   parbegin
   .  produttore; consumatore
    parend
end.
```

Figura 5.17 *Soluzione del problema del produttore/consumatore con buffer limitato utilizzando i semafori*

un solo cliente alla volta può pagare. I barbieri passano il tempo tagliando i capelli, alla cassa, o dormendo sulla sedia aspettando che arrivino clienti.

Un barbiere ingiusto

La Figura 5.20 mostra un'implementazione con semafori; le tre procedure appaiono una a fianco dell'altra per risparmiare spazio. Si suppone che tutte le code dei semafori siano del tipo first-in-first-out.

Il corpo principale del programma attiva 50 clienti, 3 barbieri e il processo cassiere. Ci sono vari operatori di sincronizzazione, ciascuno con uno scopo e una posizione precisa:

```
wait(s):
repeat ( nulla ) until testset(s.flag);
s.contatore := s.contatore - 1;
if  s.contatore < 0
    then begin
        metti questo processo in s.coda;
        blocca questo processo (bisogna anche mettere s.flag a 0)
        end
    else s.flag := 0;

signal(s):
repeat ( nulla ) until testset(s.flag);
s.contatore := s.contatore + 1;
if  s.contatore ≤ 0
    then begin
        togli un processo P da s.coda;
        metti il processo P in stato Ready
         end;
s.flag := 0;
```

(a) Istruzione Test-And-Set

```
wait(s):
disabilita le interruzioni;
s.contatore := s.contatore - 1;
if  s.contatore < 0
    then begin
        metti questo processo in s.coda;
        blocca questo processo e riattiva le interruzioni
        end
    else riattiva le interruzioni;

signal(s):
disabilita le interruzioni;
s.contatore := s.contatore + 1;
if  s.contatore ≤ 0
    then begin
        togli un processo P da s.coda;
        metti il processo P in stato Ready
         end;
riattiva le interruzioni;
```

(b) Interruzioni

Figura 5.18 *Due possibili implementazioni dei semafori*

- **Capacità del negozio e del divano:** sono governate rispettivamente dai semafori *capacita_max* e *divano*; ogni volta che un cliente cerca di entrare nel negozio, il semaforo *capacita_max* viene decrementato di un'unità, e viene incrementato ogni volta che un cliente

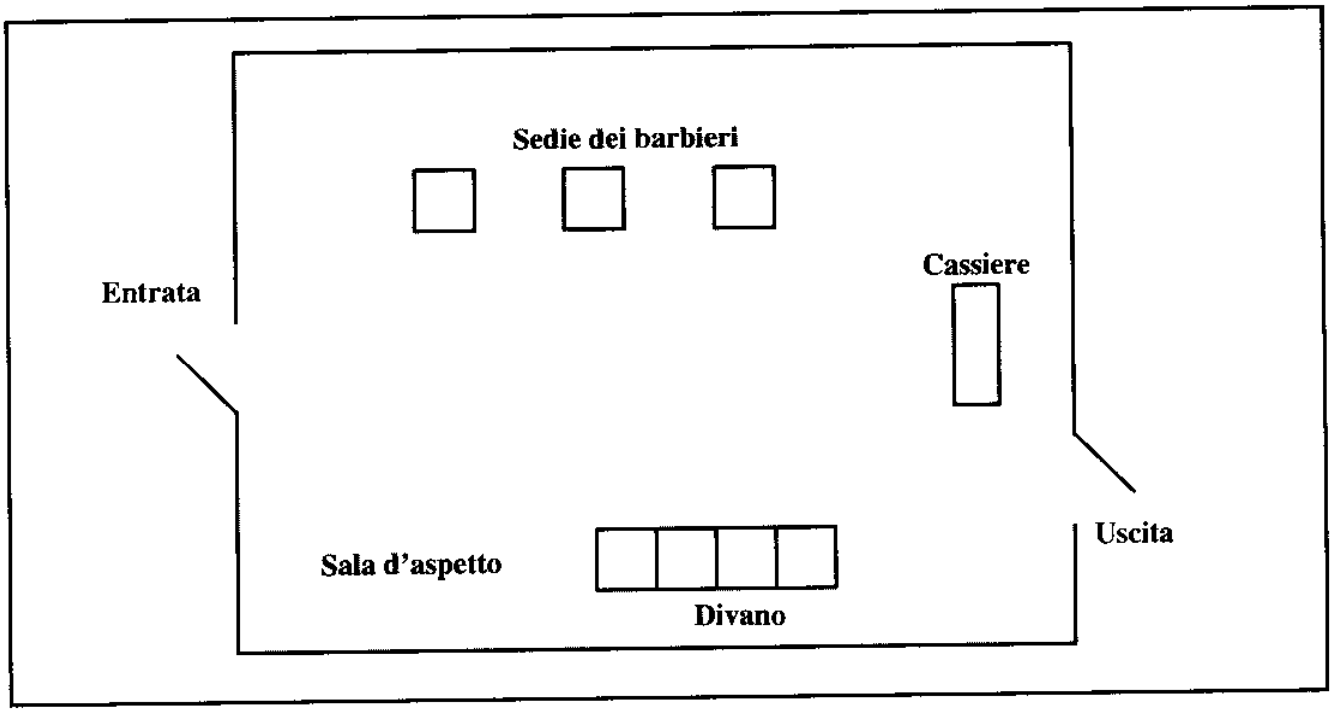


Figura 5.19 *Il negozio del barbiere*

esce. Se un cliente trova il negozio pieno, quel processo è sospeso su *capacita_max* dalla funzione *wait*. Analogamente le azioni di sedersi e alzarsi dal divano sono precedute da una *wait* e seguite da una *signal*.

- **Capacità della sedia di un barbiere:** ci sono tre sedie, e bisogna fare in modo che siano usate correttamente. Il semaforo *sedia_barbiere* assicura che non più di tre clienti alla volta chiedano di essere serviti, per evitare la circostanza poco dignitosa di un cliente seduto in braccio a un altro. Un cliente non si deve alzare dal divano a meno che non ci sia una sedia libera [*wait(sedia_barbiere)*], e ogni barbiere effettua una *signal* quando un cliente lascia la sedia [*signal(sedia_barbiere)*]. L'organizzazione a coda dei semafori garantisce un accesso equo alle sedie dei barbieri: il primo cliente che viene bloccato è anche il primo ad essere servito quando c'è una sedia disponibile. Si noti che, nella procedura che realizza il cliente, se *wait(sedia_barbiere)* fosse prima di *signal(divano)*, allora ogni cliente si sederebbe per un attimo sul divano e poi si metterebbe in coda per le sedie dei barbieri, occupando spazio e intralciando il loro lavoro.
- **Assicurarsi che ci siano i clienti sulle sedie:** il semaforo *cliente_pronto* serve per svegliare un barbiere che dorme, perché indica che un cliente si è appena seduto; senza questo semaforo un barbiere non dormirebbe mai, ma inizierebbe il taglio non appena un cliente lascia la sedia, e se nessun altro cliente prende posto il barbiere taglierebbe l'aria.
- **Tenere i clienti sulla sedia:** una volta seduto, un cliente rimane sulla sedia finché il barbiere non segnala che il taglio è finito, con il semaforo *finito*.
- **Assicurare che ci sia un solo cliente per sedia:** il semaforo *sedia_barbiere* serve per limitare a tre il numero di clienti seduti sulle sedie, comunque ciò non è sufficiente: se il cliente al termine del taglio non riesce ad avere il processore subito dopo la *signal(finito)* del corri-

```
program barbiere1;
var    capacita_max: semaforo (:= 20);
       divano: semaforo (:= 4);
       sedia_barbiere, coord: semaforo (:= 3);
       cliente_pronto, finito, lascia_sedia_b, pagamento,
       ricevuta: semaforo (:= 0);
```

```
procedure cliente;
begin
   wait(capacita_max)
   entra nel negozio;
   wait(divano);
   siedi sul divano;
   wait(sedia_barbiere)
   alzati dal divano;
   signal(divano);
   siedi sulla sedia del barbiere;
   signal(cliente_pronto);
   wait(finito);
   lascia la sedia del barbiere;
   signal (lascia_sedia_b);
   paga;
   signal(pagamento);
   wait(ricevuta);
   esci dal negozio;
   signal(capacita_max)
end;
```

```
procedure barbiere;
begin
   repeat
      wait(cliente_pronto);
      wait(coord);
      taglia i capelli;
      signal(coord);
      signal(finito);
      wait(lascia_sedia_b);
      signal(sedia_barbiere);
   forever
end;

procedure cassiere;
begin
   repeat
      wait(pagamento);
      wait(coord);
      prendi il denaro;
      signal(coord);
      signal(ricevuta);
   forever
end;
```

```
begin (* programma principale *)
    parbegin
       cliente; . . . 50 volte; . . . cliente;
       barbiere; barbiere; barbiere;
       cassiere
    parend
end.
```

Figura 5.20 *Un barbiere ingiusto*

spondente barbiere (se si addormenta, o si ferma a parlare con un vicino), allora è possibile che sia ancora sulla sedia quando il nuovo cliente viene invitato a prendere posto. Il semaforo *lascia_sedia_b* serve per correggere questo problema impedendo al barbiere di invitare un nuovo cliente finché il precedente non dichiara di volersi alzare. Nei problemi alla fine del capitolo si vedrà che anche questa precauzione non è sufficiente ad evitare la situazione spiacevole di due clienti sulla stessa sedia.

- **Pagare e incassare:** naturalmente è necessario fare attenzione al denaro: il cassiere vuole essere sicuro che ogni cliente paghi prima di lasciare il negozio, e il cliente vuole una ricevuta di pagamento. Questo viene realizzato con uno scambio di denaro faccia a faccia: ogni cliente, dopo essersi alzato dalla sedia, paga e avverte il cassiere dell'avvenuto passaggio di mano del denaro [*signal(pagamento)*], quindi aspetta la ricevuta [*wait(ricevuta)*]. Il processo cassiere non fa altro che ricevere i pagamenti: aspetta che venga segnalato un pagamento, riceve i soldi e segnala la ricezione del denaro. Ci sono molti errori di programmazione da evitare: se *signal(pagamento)* avviene subito prima dell'azione *paga*, allora il cliente che vuole pagare potrebbe venire interrotto, e ciò permetterebbe al cassiere di accettare pagamenti che in realtà non sono avvenuti. Un errore ancora più serio si avrebbe scambiando la riga di *signal(pagamento)* con quella di *wait(ricevuta)*: ciò provocherebbe uno stallo, perché tutti i clienti e il cassiere sarebbero bloccati sulle proprie operazioni *wait*.
- **Coordinare le funzioni di barbiere e cassiere:** per risparmiare, il negozio non ha assunto un cassiere, ma affida questo ruolo ai barbieri quando non sono impegnati in un taglio. Il semaforo *coord* assicura che i barbieri svolgano un solo compito alla volta.

La Tabella 5.3 riassume l'uso di ogni semaforo nel programma.

Il processo cassiere si potrebbe eliminare integrando la funzione di pagamento nella procedura del barbiere: ogni barbiere si farebbe pagare subito dopo il taglio. Comunque, poiché c'è un solo registratore di cassa, bisogna assicurarsi che venga usato da un solo barbiere alla volta, ad esempio considerando la sua funzione come una sezione critica governata da un semaforo.

Un barbiere equo

La Figura 5.20 è un buon tentativo, ma non elimina tutte le difficoltà; uno dei problemi viene risolto in questa sezione, e gli altri sono lasciati al lettore per esercizio (vedi Problema 5.19).

Nella Figura 5.20 c'è un problema di temporizzazione che potrebbe portare ad un trattamento ingiusto dei clienti. Si supponga che ci siano tre clienti seduti sulle tre sedie: i clienti verranno bloccati su *wait(finito)* e, per come sono organizzate le code, saranno liberati nello stesso ordine in cui si sono seduti. Cosa succede se uno dei barbieri è molto veloce, o uno dei clienti ha pochi capelli? Liberare il cliente che è entrato nella coda per primo provocherebbe una situazione in cui un cliente deve lasciare la propria sedia e pagare per un taglio lasciato a metà, mentre un altro è forzato a rimanere seduto anche se il suo taglio è finito.

Il problema viene risolto usando più semafori, come mostrato nella Figura 5.21: si assegna un numero unico per ogni cliente, che è equivalente a dare un numero ai clienti quando entrano nel negozio. Il semaforo *mutex1* protegge l'accesso alla variabile globale *contatore*, così ogni cliente riceve un numero diverso. Il semaforo *finito* viene ridefinito come un array di 50 sema-

```
program barbiere2;
var    capacita_max: semaforo (:= 20);
       divano: semaforo (:= 4);
       sedia_barbiere, coord: semaforo (:= 3);
       mutex1, mutex2: semaforo (:= 1);
       cliente_pronto, lascia_sedia_b, pagamento, ricevuta: semaforo (:= 0);
       finito: array [1..50] of semaforo (:= 0);
       contatore: integer;

  procedure cliente;
  var numcli: integer;
  begin
     wait(capacita_max)
     entra nel negozio;
     wait(mutex1);
     contatore := contatore + 1;
     numcli := contatore;
     signal(mutex1);
     wait(divano);
     siedi sul divano;
     wait(sedia_barbiere)
     alzati dal divano;
     signal(divano);
     siedi sulla sedia del barbiere;
     wait(mutex2);
     incoda1(numcli);
     signal(cliente_pronto);
     signal(mutex2);
     wait(finito[numcli]);
     lascia la sedia del barbiere;
     signal (lascia_sedia_b);
     paga;
     signal(pagamento);
     wait(ricevuta);
     esci dal negozio;
     signal(capacita_max)
  end;

  procedure barbiere;
  var cliente_b: integer;
  begin
     repeat
        wait(cliente_pronto);
        wait(mutex2);
        decoda1(cliente_b);
        signal(mutex2);
        wait(coord);
        taglia i capelli;
        signal(coord);
        signal(finito[cliente_b]);
        wait(lascia_sedia_b);
        signal(sedia_barbiere);
     forever
  end;

  procedure cassiere;
  begin
     repeat
        wait(pagamento);
        wait(coord);
        prendi il denaro;
        signal(coord);
        signal(ricevuta);
     forever
  end;

  begin (* programma principale *)
     contatore := 0;
      parbegin
        cliente; . . . 50 volte; . . . cliente;
        barbiere; barbiere; barbiere;
        cassiere
     parend
  end.
```

Figura 5.21 *Un barbiere equo*

nel caso di una lista, si può mettere un unico lock per tutte le liste, oppure un lock diverso per ogni lista, o ancora un lock per ogni elemento della lista.

Per iniziare si analizza la versione di Hoare e in seguito una più raffinata.

Monitor con segnali

Un monitor è un modulo software che contiene una o più procedure, una sequenza di inizializzazione e dati locali; le caratteristiche principali sono le seguenti:

1. Le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne.
2. Un processo entra nel monitor chiamando una delle sue procedure.
3. Solo un processo alla volta può essere in esecuzione all'interno del monitor; ogni altro processo che ha chiamato il monitor è sospeso, nell'attesa che questo diventi disponibile.

Le prime due caratteristiche ricordano quelle degli oggetti nel software orientato agli oggetti, infatti i sistemi operativi e i linguaggi orientati agli oggetti possono implementare un monitor come un oggetto con caratteristiche speciali.

Garantendo l'esecuzione di un solo processo alla volta, un monitor può essere usato per la mutua esclusione; poiché le variabili locali sono accessibili da un solo processo alla volta, si possono proteggere le strutture dati condivise semplicemente mettendole all'interno del monitor. Se i dati di un monitor rappresentano una risorsa, allora il monitor fornisce la mutua esclusione per l'accesso a quella risorsa.

Per essere utile nell'elaborazione concorrente, un monitor deve contenere degli strumenti di sincronizzazione. Ad esempio, si supponga che un processo chiami il monitor, e che, mentre è al suo interno, sia sospeso finché non si verifica una certa condizione. È necessario fare in modo che il processo sospeso rilasci il monitor, in modo che altri processi possano entrare. Quando in seguito la condizione viene soddisfatta e il monitor è nuovamente disponibile, il processo deve essere riattivato e deve poter rientrare nel monitor nello stesso punto in cui era stato sospeso.

Un monitor fornisce la sincronizzazione mediante l'uso di *variabili di condizione*, accessibili solo dall'interno del monitor. Due funzioni operano sulle variabili di condizione:

cwait(c): sospende l'esecuzione del processo chiamante sulla condizione *c*; il monitor diventa disponibile per gli altri processi.

csignal(c): riattiva un processo sospeso sulla condizione *c*; se i processi sospesi sono molti, ne sceglie uno; se non ce ne sono, non fa niente.

Si noti che le operazioni *cwait/csignal* dei monitor sono diverse da quelle dei semafori: se un processo in un monitor effettua una signal e non c'è nessun processo bloccato su quella variabile di condizione, il segnale viene perso.

La Figura 5.22 illustra la struttura di un monitor: sebbene un processo possa entrare nel monitor chiamando una delle sue procedure, si può pensare che il monitor abbia una sola entrata, che viene controllata in modo che un solo processo alla volta possa entrare, e che gli altri

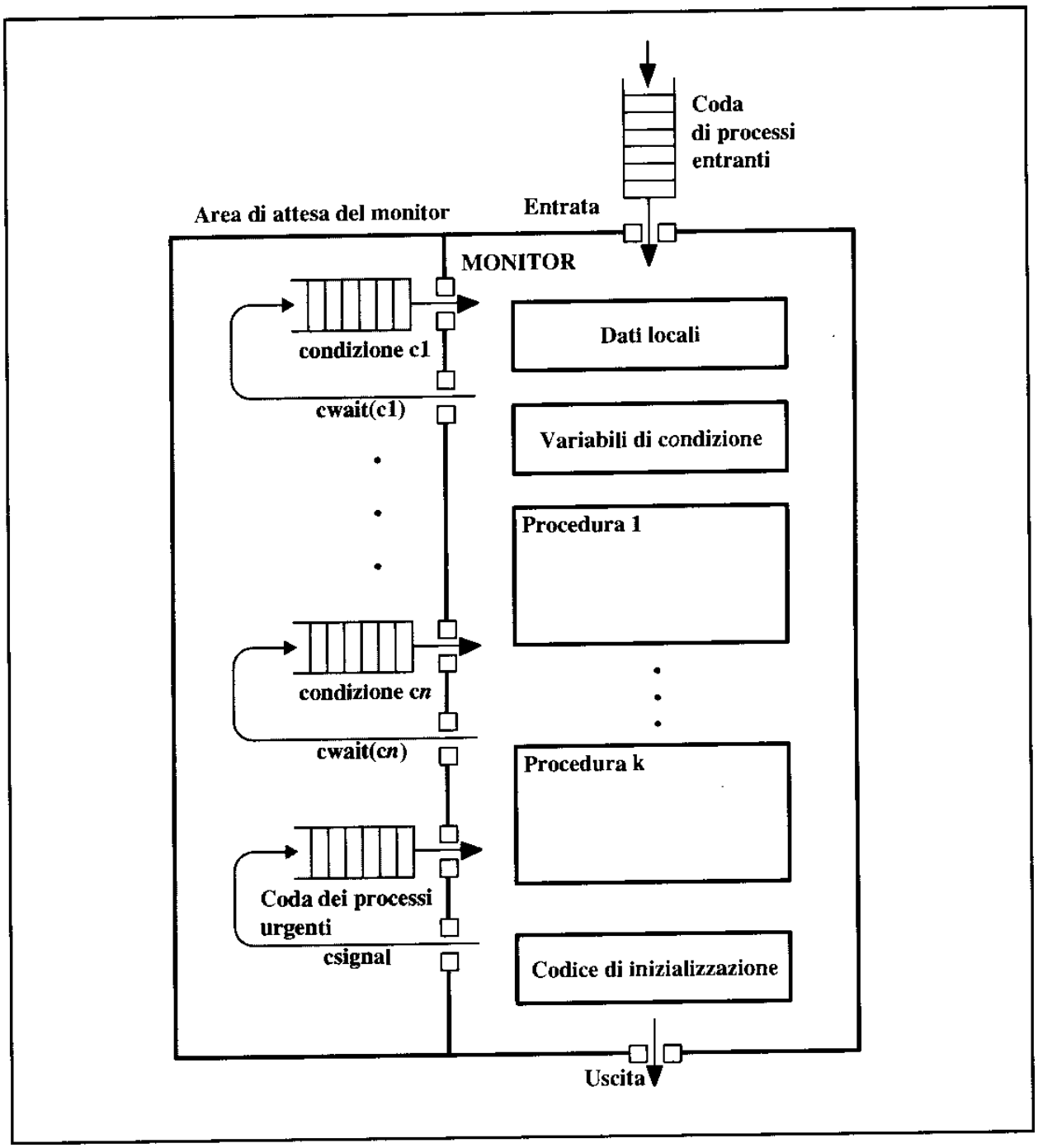


Figura 5.22 *Struttura di un monitor*

processi che vogliono entrare formino la coda dei processi sospesi in attesa della disponibilità del monitor. Quando un processo è nel monitor, può sospendersi sulla condizione x con una *cwait(x)*: viene messo in una coda di processi che aspettano di rientrare nel monitor quando la condizione cambia.

Se un processo in esecuzione all'interno di un monitor rileva il cambiamento di una variabile di condizione x, manda un *csignal(x)* per avvisare la coda corrispondente che c'è stato un cambiamento.

Per fare un esempio di uso dei monitor, si consideri nuovamente il problema del produttore/consumatore con buffer limitato. La Figura 5.23 mostra una soluzione usando un monitor: il modulo del monitor, *bufferlimitato*, controlla il buffer usato per contenere i caratteri. Il monitor contiene due variabili di condizione: *nonpieno* è vera quando nel buffer c'è spazio per almeno un altro carattere, e *nonvuoto* quando c'è almeno un carattere nel buffer.

Un produttore può aggiungere caratteri al buffer solo utilizzando la procedura *aggiungi* dall'interno del monitor, e non ha accesso diretto alla variabile *buffer*. La procedura per prima cosa controlla la condizione *nonpieno*, per vedere se c'è spazio disponibile e in caso negativo sospende su quella condizione il processo in esecuzione all'interno del monitor, così un altro processo (produttore o consumatore) può entrare. Quando in seguito il buffer non è più pieno, il processo sospeso può essere rimosso dalla coda, riattivato, e può riprendere l'esecuzione. Dopo aver messo un carattere nel buffer, il processo segnala la condizione *nonvuoto*. Il consumatore si comporta in maniera simile.

Quest'esempio sottolinea la differenza fra la divisione delle responsabilità nei monitor e nei semafori: i monitor di per sé garantiscono la mutua esclusione, infatti non è possibile che un produttore e un consumatore accedano contemporaneamente al buffer; comunque il programmatore deve mettere *cwait* e *csignal* nel punto giusto per evitare che i processi tentino di aggiungere elementi al buffer pieno o di prelevarne da uno vuoto. Nel caso dei semafori, sia la mutua esclusione sia la sincronizzazione sono a carico del programmatore.

Si noti che nella Figura 5.23 un processo esce dal monitor subito dopo aver eseguito *csignal*; se *csignal* comparisse altrove, non alla fine della procedura, allora, nella proposta di Hoare, il processo che effettua *csignal* viene sospeso per rendere il monitor disponibile, e messo in una coda finché il monitor non si libera. A questo punto si potrebbe mettere il processo nella coda di entrata, così dovrebbe competere per l'accesso con gli altri processi che non sono ancora entrati nel monitor. Poiché però un processo sospeso su una *csignal* ha già effettuato parte del lavoro nel monitor, è sensato dargli la precedenza rispetto ai processi che cercano di entrare per la prima volta, creando una nuova coda per i processi urgenti (Figura 5.22). Concurrent Pascal, un linguaggio che usa i monitor, richiede che *csignal* compaia solo come ultima operazione delle procedure del monitor.

Se non ci sono processi in attesa su una condizione *x*, l'esecuzione di *csignal(x)* non produce alcun effetto.

Come con i semafori, è possibile commettere errori nella scrittura delle funzioni di sincronizzazione del monitor: ad esempio, se si elimina una delle funzioni *csignal* dal monitor per il buffer limitato, i processi che vengono messi nelle code delle variabili di condizione rimangono bloccati per sempre. Il vantaggio dei monitor rispetto ai semafori è che tutte le funzioni di sincronizzazione sono raccolte nel monitor, quindi è più facile verificare la correttezza della sincronizzazione e individuare gli errori. Inoltre, quando un monitor è stato scritto correttamente, l'accesso alle risorse condivise da parte dei processi sarà sempre corretto. Al contrario, con i semafori l'accesso alle risorse è corretto solo se tutti i processi che accedono alle risorse sono programmati correttamente.

Monitor con notifica e diffusione

La definizione di monitor data da Hoare [HOAR74] prevede che, se c'è almeno un processo

```
program produttore_consumatore
monitor bufferlimitato;
     buffer: array [0.. N-1] of char;        {spazio per N elementi }
     nextin, nextout: integer;               {puntatori al buffer }
     contatore : integer;                    {numero di elementi nel buffer}
     nonpieno, nonvuoto: condition;          {per la sincronizzazione }

procedure aggiungi (x: char);
begin
    if contatore = N then cwait(nonpieno);   {il buffer è pieno; evita
                                              l'overflow }
   buffer[nextin] := x;
   nextin := nextin + 1 mod N;
   contatore := contatore + 1;               {un elemento in più nel buffer}
   csignal(nonvuoto);                        {sveglia i consumatori in attesa}
 end;
procedure togli (x: char);
begin
    if contatore = 0 then cwait(nonvuoto);   {il buffer è pieno; evita
                                              l'underflow}
   x := buffer[nextout];
   nextout := nextout + 1 mod N;
   contatore := contatore - 1;               {un elemento in meno nel buffer}
   csignal(nonpieno);                        {sveglia i produttori in attesa}
end;
begin                                        {monitor}
   nextin := 0; nextout := 0; contatore := 0 {buffer inizialmente vuoto}
end;

procedure produttore;
var   x: char;
begin
   repeat
      produci(x);
      aggiungi (x);
   forever
end;
procedure consumatore;
var  x: char;
begin
   repeat
      togli(x);
      consuma(x);
   forever
end;
begin (* programma principale *)
   parbegin
      produttore; consumatore
   parend
end.
```

Figura 5.23 *Soluzione al problema del produttore/consumatore con buffer limitato usando un monitor*

in una coda di condizione, un processo che era in tale coda venga attivato immediatamente, quando un altro processo effettua la *csignal* per quella condizione. Pertanto, il processo che effettua la *signal* deve uscire immediatamente dal monitor o restare sospeso.

Quest'approccio presenta degli svantaggi:

1. Se il processo che effettua la *csignal* non ha terminato la sua azione all'interno del monitor, allora sono necessari due scambi di contesto fra processi: uno per sospenderlo e uno per riattivarlo quando il monitor diventa disponibile.
2. La schedulazione dei processi associata ad una *csignal* deve essere completamente affidabile: quando viene effettuata una *csignal*, bisogna riattivare immediatamente un processo della coda di condizione corrispondente, e lo schedulatore deve garantire che nel frattempo nessun altro processo entri nel monitor (altrimenti la condizione su cui il processo è attivato potrebbe cambiare). Ad esempio, nella Figura 5.23, quando si effettua una *csignal(nonvuoto)*, bisogna riattivare un processo della coda *nonvuoto* prima che un nuovo consumatore entri nel monitor. Un altro esempio è quello di un processo produttore, che aggiunge un carattere al buffer e fallisce prima della *csignal*: ogni processo nella coda *nonvuoto* è bloccato per sempre.

Lampson e Redell hanno sviluppato una definizione diversa di monitor per il linguaggio Mesa [LAMP80]; il loro approccio supera i problemi appena visti e fornisce molte estensioni utili. La struttura dei monitor di Mesa è usata anche nel linguaggio di programmazione Modula-3 [CARD89,HARB90,NELS91]. In Mesa la primitiva *csignal* è rimpiazzata da *cnotify*, con l'interpretazione seguente: quando un processo in esecuzione nel monitor esegue *cnotify(x)*, esso manda un segnale alla coda corrispondente a *x*, ma il processo non viene sospeso; il risultato è che il processo in cima alla coda sarà riattivato successivamente, in un momento opportuno, quando il monitor è disponibile. Comunque, poiché non c'è garanzia che nessun altro processo entri nel monitor prima del processo in attesa, questo dovrà controllare nuovamente la condizione. Ad esempio le procedure del monitor *buffer_limitato* andrebbero riscritte in questo modo:

```
procedure aggiungi (x: char);
  begin
      while contatore = N do cwait(nonpieno);
                {il buffer è pieno; evita l'overflow}
      buffer[nextin] := x;
      nextin := nextin + 1 mod N;
      contatore := contatore + 1;  {un elemento in più
                                     nel buffer}
      cnotify{nonvuoto};  {avverte i consumatori in
                           attesa}
  end;
procedure togli (x: char);
  begin
      while contatore = 0 do cwait(nonvuoto);
              {il buffer è vuoto; evita l'underflow}
      x := buffer[nextout];
```

```
        nextout:= nextout + 1 mod N;
        contatore := contatore - 1;  {un elemento in meno
                                      nel buffer}
        cnotify(nonpieno);       {avverte i produttori
                                 in attesa}
    end;
```

I comandi **if** sono sostituiti con cicli **while**, così si effettua almeno una valutazione della variabile di condizione in più di prima, ma nello stesso tempo non si effettuano scambi di contesto fra processi inutili, e non ci sono vincoli su quando il processo in attesa deve essere riattivato dopo una *cnotify*.

Un utile raffinamento associato alla primitiva *cnotify* è effettuare un controllo sul tempo associato ad ogni condizione: un processo che ha aspettato per il tempo massimo stabilito, è messo in stato Ready indipendentemente dalla condizione. Quando un processo è attivato, controlla la condizione e continua se questa è soddisfatta. Il limite di tempo previene l'attesa infinita di un processo nel caso che un altro processo fallisca prima di poter segnalare una condizione.

Con la regola che un processo riceve una notifica invece di essere riattivato immediatamente, è possibile aggiungere una primitiva *cbroadcast*: questa ha l'effetto di mettere in stato Ready tutti i processi in attesa su una condizione. Ciò è utile nelle situazioni in cui un processo non sa quanti processi devono essere riattivati. Ad esempio, nel programma del produttore/consumatore, si supponga che le funzioni per aggiungere e togliere elementi dal buffer gestiscano blocchi di caratteri di dimensione variabile: se un produttore aggiunge un blocco di caratteri al buffer, non è detto che conosca il numero di caratteri che ciascun consumatore in attesa vuole prelevare, così effettua una *cbroadcast* che avvisa tutti i processi in attesa.

L'operazione *cbroadcast* è utile anche quando un processo non può stabilire quali sono i processi da riattivare; un buon esempio è quello di un gestore della memoria: il gestore ha j byte liberi, e un processo libera altri k byte ma non sa quali processi in attesa possono essere soddisfatti con $k+j$ byte, quindi effettua un cbroadcast e ogni processo controlla se la memoria libera è sufficiente per sé.

Un vantaggio dei monitor di Lampson/Redell rispetto a quelli di Hoare è che usando i primi si tende a commettere meno errori. Nell'approccio di Lampson/Redell, poiché ogni procedura controlla la variabile di condizione quando riceve il segnale, un processo può, tramite l'uso del costrutto **while**, effettuare una *csignal* o una *cbroadcast* non corretta senza causare errori nei processi che la ricevono, perché questi ricontrolleranno la variabile in questione, e continueranno ad aspettare se la condizione desiderata non sarà verificata.

Un altro vantaggio dei monitor di Lampson/Redell è che portano ad un approccio più modulare alla costruzione dei programmi. Ad esempio, si consideri l'implementazione di un allocatore di buffer; ci sono due livelli di condizioni da soddisfare per la cooperazione dei processi sequenziali:

1. Strutture dati coerenti: il monitor garantisce la mutua esclusione e completa un'operazione d'input/output prima di permettere un'altra operazione sul buffer.
2. Il livello 1, più abbastanza memoria per permettere al processo di completare la richiesta d'allocazione.

Nei monitor di Hoare ogni segnale garantisce il livello 1, e in aggiunta porta il messaggio implicito: "Ho liberato spazio sufficiente per la tua richiesta d'allocazione". Così il segnale garantisce implicitamente anche la condizione del livello 2. Se in seguito il programmatore cambia la definizione della condizione del livello 2, sarà necessario riprogrammare tutti i processi che mandano segnali. Se il programmatore cambia le supposizioni fatte da un processo in attesa (ad esempio se è in attesa su un diverso invariante del livello 2), può essere necessario riprogrammare tutti i processi che mandano segnali. Ciò non è modulare, e causa facilmente degli errori di sincronizzazione (ad esempio riattivazione per errore) quando il codice è modificato. Il programmatore deve ricordarsi di modificare tutti i processi ogni volta che fa un piccolo cambiamento alla condizione del livello 2. Con i monitor di Lampson/Redell, al contrario, la *cbroadcast* garantisce la soddisfazione della condizione di livello 1, e implicitamente suggerisce che anche la condizione di livello 2 potrebbe essere valida; ogni processo deve controllare la validità della condizione di livello 2. Se una condizione di livello 2 è cambiata da una *cwait* o *cnotify*, non è possibile che un qualche processo sia riattivato per errore, perché ogni procedura deve controllare la condizione, quindi la condizione di livello 2 si può nascondere all'interno della procedura. Nei monitor di Hoare, le condizioni di livello 2 sono spostate dal processo in attesa al codice d'ogni processo segnalante, il che viola i principi d'astrazione dai dati e modularità fra procedure.

5.6 Scambio di messaggi

Quando i processi interagiscono fra loro, devono soddisfare due requisiti fondamentali: devono essere sincronizzati per garantire la mutua esclusione, e hanno bisogno di scambiarsi informazioni per cooperare. Un modo per soddisfare entrambi è lo scambio di messaggi, che in più ha il vantaggio di un'implementazione naturale sui sistemi distribuiti, ma anche sui multiprocessori con memoria condivisa e sui sistemi a singolo processore.

I sistemi a scambio di messaggi sono di diversi tipi; in questa sezione si farà un'introduzione generale alle caratteristiche tipiche di questi sistemi. Solitamente il supporto allo scambio di messaggi è dato da due primitive:

```
send(destinazione, messaggio)
receive(sorgente, messaggio)
```

Questo è il numero minimo d'operazioni necessarie ai processi per scambiarsi messaggi. Un processo manda informazioni in forma di *messaggio* ad un altro processo destinatario. Un processo riceve informazioni eseguendo la primitiva *receive*, indicando il mittente e il *messaggio*.

La Tabella 5.4 elenca vari criteri di progettazione dei sistemi a scambio di messaggi, che saranno discussi in dettaglio nel resto di questa sezione.

Sincronizzazione

Per scambiare un messaggio, i due processi coinvolti devono in parte sincronizzarsi: il rice-

Tabella 5.4 *Caratteristiche di progettazione dei sistemi di messaggi per comunicazione interprocessore e sincronizzazione*

Sincronizzazione	**Formato**
Send	Contenuto
bloccante	Lunghezza
non bloccante	fissata
Receive	variabile
bloccante	
non bloccante	**Tipo di Coda**
test di messaggi in arrivo	FIFO
Indirizzamento	A priorità
Diretto	
send	
receive	
esplicito	
implicito	
Indiretto	
statico	
dinamico	
possesso	

vente non può acquisire un messaggio prima che il mittente lo abbia inviato; occorre inoltre precisare cosa accade ad un processo che chiama la primitiva *send* o *receive*.

Si consideri la primitiva *send*: quando è eseguita ci sono due possibilità, a seconda che il processo mittente sia o non sia bloccato fino alla ricezione del messaggio; analogamente, ci sono due possibilità quando un processo effettua una *receive:*

1. Se il messaggio era già stato mandato, il processo lo riceve e continua l'esecuzione.
2. Se non ci sono messaggi in arrivo, allora si verifica uno dei due casi seguenti: (a) il processo è bloccato fino all'arrivo di un messaggio, oppure (b) continua l'esecuzione, rinunciando a ricevere un messaggio.

Quindi la *send* e la *receive* possono essere bloccanti o non bloccanti; le combinazioni più comuni sono tre, anche se ogni sistema reale implementa una o al massimo due combinazioni:

- **Send bloccante e receive bloccante:** sia il mittente sia il ricevente sono bloccati fino al completamento dell'operazione: spesso questo è detto *rendez-vous*. Tale combinazione permette una sincronizzazione stretta fra processi.
- **Send non bloccante e receive bloccante:** anche se il mittente può continuare, il ricevente deve aspettare l'arrivo del messaggio; questa è probabilmente la combinazione più utile, perché permette ad un processo di mandare vari messaggi a diversi destinatari molto velocemente. Quando un processo ha bisogno di ricevere un messaggio per continuare, è necessario bloccarlo finché il messaggio non arriva. Un esempio è un processo server che assolve il compito di fornire un servizio o una risorsa agli altri processi.

- **Send non bloccante e receive non bloccante:** nessuno dei due deve aspettare.

In molti casi la *send* non bloccante è la più naturale per la programmazione concorrente; ad esempio, se è usata per chiamare un'operazione di output, come la stampa, consente al processo di mandare la richiesta in forma di messaggio e continuare l'esecuzione. Un pericolo potenziale della *send* non bloccante è che un errore potrebbe portare ad una situazione in cui un processo continua a generare messaggi; siccome non c'è modo di bloccare il mittente, i messaggi possono consumare le risorse di sistema, in particolare il tempo di processore e lo spazio dei buffer, a danno degli altri processi e del sistema operativo. Inoltre, la *send* non bloccante lascia al programmatore il compito di controllare che un messaggio arrivi a destinazione: i processi devono mandare messaggi di risposta per confermare la ricezione di un messaggio.

Per la primitiva *receive*, la versione bloccante sembra la più naturale per molti casi di programmazione concorrente. In genere un processo che richiede un messaggio avrà bisogno, per procedere, delle informazioni in esso contenute; tuttavia, se un messaggio è perso, cosa che può accadere nei sistemi distribuiti, o se un processo fallisce prima di poter mandare il messaggio, il processo ricevente rimarrà bloccato per sempre. Il problema si può risolvere utilizzando una *receive* non bloccante; in questo secondo caso il pericolo è che ogni messaggio, inviato dopo che il processo ha già effettuato la *receive* corrispondente, andrà perso. Un altro approccio possibile è dare al processo la possibilità di controllare se c'è un messaggio in arrivo prima di effettuare la *receive*; oppure specificare più di un mittente in una *receive*. Quest'ultima possibilità è utile quando un processo aspetta messaggi provenienti da processi diversi, e se un solo messaggio è sufficiente per continuare l'esecuzione.

Indirizzamento

Chiaramente la primitiva *send* richiede di specificare quale processo deve ricevere il messaggio; analogamente molte implementazioni permettono al processo ricevente di specificare il mittente del messaggio da ricevere.

Gli schemi per specificare i processi nelle *send* e *receive* si dividono in due categorie: indirizzamento diretto e indiretto. Con l'**indirizzamento diretto** la primitiva *send* contiene un identificatore specifico del processo destinatario. La primitiva *receive* si può gestire in due modi. Una possibilità è di richiedere di indicare esplicitamente il processo mittente, così il processo deve sapere in anticipo qual è il processo da cui aspetta un messaggio; questo è fattibile per i processi concorrenti che cooperano. In altri casi, comunque, è impossibile prevedere il mittente del messaggio: ad esempio un processo server per la stampante deve accettare richieste di stampa da qualunque processo. Per tali applicazioni, un approccio più efficace è l'uso dell'indirizzamento implicito: il parametro *sorgente* della primitiva *receive* serve per rispondere al mittente del messaggio quando l'operazione è compiuta.

L'altro approccio è l'**indirizzamento indiretto**: i messaggi non viaggiano direttamente dal mittente al destinatario ma sono mandati ad una struttura dati condivisa che si compone di code che contengono temporaneamente i messaggi. In genere tali code sono dette caselle della posta; per portare a termine la comunicazione, un processo manda un messaggio alla casella appropriata, e l'altro processo prende il messaggio da quella casella.

Un vantaggio dell'uso dell'indirizzamento indiretto è che, separando il mittente e il riceven-

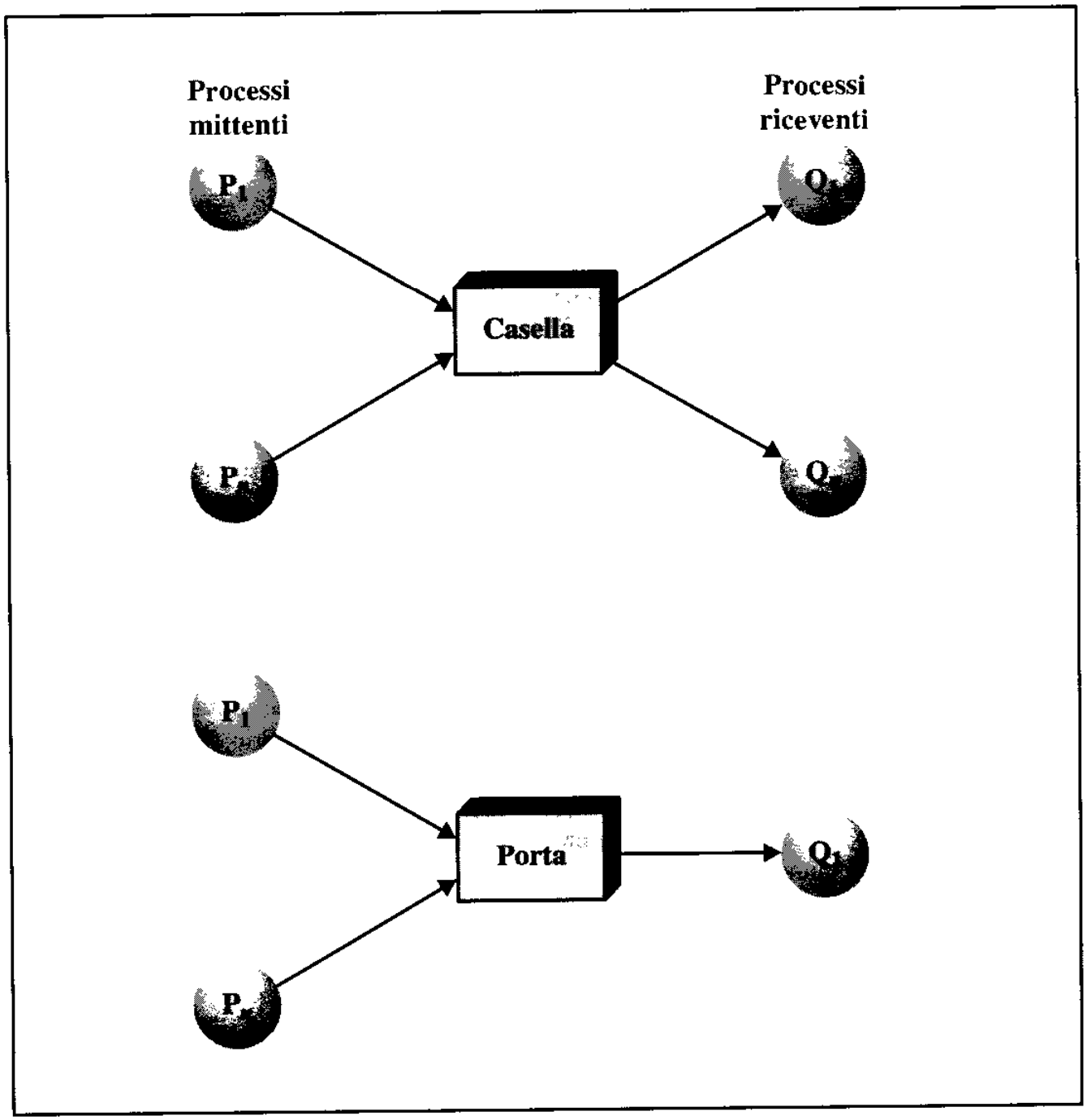


Figura 5.24 *Comunicazione indiretta fra processi*

te, permette una maggiore flessibilità nell'uso dei messaggi. La relazione fra mittenti e riceventi può essere uno-a-uno, molti-a-uno, uno-a-molti, o molti-a-molti. Una relazione uno-a-uno permette l'uso di connessioni private per la comunicazione fra due processi: in questo modo s'isolano le interazioni fra i due da possibili interferenze causate da errori degli altri processi. Una relazione molti-a-uno è utile per l'interazione client/server: un processo fornisce un servizio per molti processi, e in questo caso spesso la casella della posta è detta porta (Figura 5.24). Una relazione uno-a-molti permette di avere un mittente e molti destinatari: è utile per applicazioni dove un messaggio o qualche informazione deve essere mandata ad un insieme di processi.

L'associazione dei processi alle caselle di posta può essere statica o dinamica. Spesso le

porte sono associate staticamente ad un processo particolare, cioè la porta è creata e assegnata permanentemente al processo. Analogamente, una relazione uno-a-uno viene di solito definita staticamente e permanentemente. Quando ci sono molti mittenti, l'associazione di un mittente ad una casella può avvenire dinamicamente, tramite l'uso di primitive come *connect* e *disconnect*.

Un argomento correlato è quello del possesso di una casella di posta; nel caso di una porta, questa di solito appartiene al processo ricevente, che è anche quello che la crea, quindi quando il processo è distrutto anche la porta è distrutta. Nel caso generale della casella di posta, il sistema operativo può offrire un servizio per creare le caselle. Tali caselle possono appartenere al processo che le crea (e così terminano insieme con il processo), oppure appartenere al sistema operativo, nel qual caso è necessario usare un comando particolare per distruggerle.

Formato dei messaggi

Il formato dei messaggi dipende dagli obiettivi del sistema di scambio di messaggi e dall'uso di un singolo computer o di un sistema distribuito. Per alcuni sistemi operativi i progettisti hanno scelto messaggi corti di lunghezza fissata, in modo da minimizzare il sovraccarico e lo spazio richiesto. Se è necessario passare una gran quantità di dati, si possono mettere i dati in un file e indicare semplicemente il nome del file nel messaggio. L'uso di messaggi di dimensione variabile costituisce un approccio più flessibile.

La Figura 5.25 mostra il formato tipico dei messaggi a lunghezza variabile forniti da un sistema operativo. Il messaggio è diviso in due parti: un'intestazione, che contiene informazioni riguardo al messaggio, e un corpo, che contiene il messaggio vero e proprio. L'intestazione può contenere un identificatore del mittente e del destinatario del messaggio, un campo per la lunghezza e un campo che indica il tipo di messaggio; eventualmente ci possono essere delle informazioni di controllo, come un puntatore per creare una lista di messaggi, un numero sequenziale per tenere traccia del numero e dell'ordine dei messaggi mandati dal mittente al destinatario, e un campo per la priorità.

Organizzazione delle code

L'organizzazione più semplice è quella first-in-first-out, ma non è sufficiente se alcuni messaggi sono più urgenti di altri. In alternativa, si può permettere di specificare la priorità dei messaggi, sulla base del tipo o su richiesta del mittente; oppure, permettere al ricevente di esaminare la coda e decidere quale messaggio ricevere per primo.

Mutua esclusione

La Figura 5.26 mostra uno dei modi di usare lo scambio di messaggi per garantire la mutua esclusione (si confrontino le Figure 5.1, 5.7 e 5.10). S'ipotizza l'uso di una *receive* bloccante e una *send* non bloccante. Un gruppo di processi concorrenti condivide una casella della posta, *mutex*, che è usata normalmente dai processi per mandare e ricevere; la casella è inizializzata con un singolo messaggio vuoto. Un processo che vuole entrare nella sezione critica prima cerca di ricevere un messaggio; se la casella è vuota, il processo è bloccato. Quando un processo ha ricevuto il messaggio, entra nella sezione critica e al termine rimanda il messaggio alla casella,

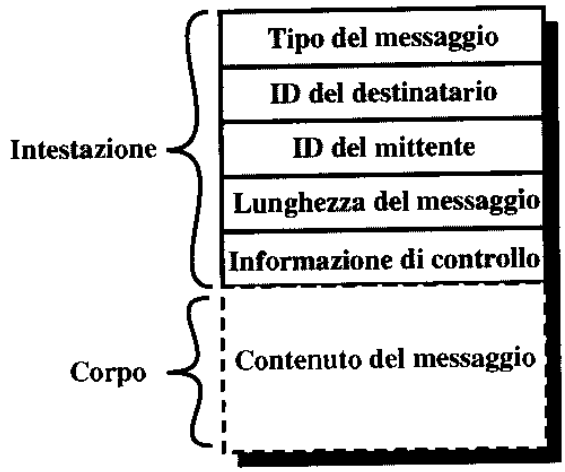


Figura 5.25 *Formato generale di un messaggio*

```
program  mutuaesclusione;
const  n =  ... ; (* numero di processi *);
procedure P(i: integer);
var msg: messaggio;
begin
   repeat
     receive(mutex, msg);
       < sezione critica >;
      send(mutex, msg);
      < resto del programma >
   forever
 end;
begin (*  programma principale *)
   crea_casella(mutex);
   send(mutex, null);
   parbegin
     P(1);
     P(2);
     ...
     P(n)
   parend
end.
```

Figura 5.26 *Mutua esclusione usando i messaggi*

così il messaggio rappresenta un permesso di accesso (*token*) che è passato da un processo all'altro.

La soluzione precedente ipotizza che, se più di un processo esegue la *receive*, allora

- Se c'è un messaggio, questo è consegnato ad un solo processo e gli altri sono bloccati.
- Se la coda di messaggi è vuota, tutti i processi sono bloccati, e quando arriva un messaggio, un solo processo è attivato e riceve il messaggio.

```
const
      capacita = ...;     { capacità del buffer }
      null =  ...; { messaggio vuoto }
var   i: integer;
procedure produttore;
var pmsg: messaggio;
begin
   while true do
      begin
         receive (cas_produci, pmsg);
         pmsg := produci;
         send (cas_consuma, pmsg)
      end
end;
procedure consumatore;
var cmsg: messaggio;
begin
   while true do
      begin
         receive (cas_consuma, cmsg);
         consuma (cmsg);
         send (cas_produci, null)
      end
end;

{processo genitore}
begin
   crea_casella (cas_produci);
   crea_casella (cas_consuma);
   for i = 1 to capacita do send (cas_produci, null);
   parbegin
      produttore;
      consumatore
   parend
end
```

Figura 5.27 *Soluzione del problema del produttore/consumatore con buffer limitato usando i messaggi*

Quest'ipotesi vale praticamente per tutti i sistemi con scambio di messaggi.

Un altro esempio dell'uso dello scambio di messaggi, in Figura 5.27, è una soluzione del problema del produttore/consumatore con buffer limitato: utilizzando le capacità di mutua esclusione dello scambio di messaggi, il problema si potrebbe risolvere con un algoritmo simile a quello della Figura 5.17; invece, il programma di Figura 5.27 sfrutta lo scambio di messaggi per passare dati insieme con i segnali. Si usano due caselle: quando il produttore genera dei dati, li manda alla casella *cas_consuma* tramite dei messaggi, e finché c'è un messaggio in quella casella, il consumatore può consumare. Quindi *cas_consuma* serve da buffer, e i dati del buffer sono organizzati come coda di messaggi; la dimensione del buffer è data dalla variabile globale *capacita*. Inizialmente la casella *cas_produci* contiene un numero di messaggi vuoti pari alla capacità del buffer; il numero diminuisce per ogni dato prodotto e aumenta per ogni dato consumato.

Quest'approccio è piuttosto flessibile: ci possono essere molti produttori e consumatori, a patto che tutti abbiano accesso ad entrambe le caselle. Il sistema può anche essere distribuito, con tutti i produttori e la casella *cas_produci* in un sito, e tutti i consumatori e la casella *cas_consuma* in un altro.

5.7 Il problema dei lettori/scrittori

Lavorando alla progettazione dei meccanismi per la concorrenza e sincronizzazione, è utile poter paragonare i problemi incontrati con problemi noti, e confrontare le soluzioni in base alla capacità di risolvere questi problemi noti; perciò in letteratura alcuni problemi hanno assunto un'importanza particolare ed appaiono sovente, sia perché sono esempi di problemi di progettazione comuni, sia per il loro valore didattico. Uno di questi problemi è quello del produttore/consumatore, che è stato già trattato. In questa sezione si considera un altro problema classico: il problema dei lettori/scrittori.

Il problema ha la seguente definizione: c'è un'area di dati condivisi fra vari processi, questa può essere un file, un blocco di memoria, o un banco di registri del processore. Ci sono dei processori che possono solo leggere i dati (lettori), e altri che possono solo scrivere (scrittori). Le condizioni seguenti devono essere soddisfatte:

1. Più lettori possono leggere il file contemporaneamente.
2. Solo uno scrittore alla volta può scrivere nel file.
3. Se uno scrittore sta scrivendo nel file, nessun lettore può leggerlo.

Prima di procedere, è necessario distinguere il problema dagli altri due finora studiati: il problema generale della mutua esclusione e il problema dei produttori/consumatori. Nel problema dei lettori/scrittori, i lettori non possono essere anche scrittori, e gli scrittori non possono leggere; il caso più generale, che contiene questo, è quello in cui tutti i processi possono leggere e scrivere: in tal caso si può dichiarare ogni porzione dei processi in cui si accede ai dati come sezione critica, e usare la soluzione generale per la mutua esclusione. La ragione per cui ci si

occupa del caso particolare è che si possono trovare delle soluzioni più efficienti, e che le soluzioni al problema generale sono di una lentezza inaccettabile in questo caso. Ad esempio, si supponga che l'area condivisa sia il catalogo di una biblioteca; gli utenti normali leggono il catalogo per cercare un libro, e un paio di bibliotecari possono aggiornare il catalogo. Nella soluzione generale, ogni accesso al catalogo sarebbe trattato come sezione critica, e gli utenti sarebbero forzati a leggere il catalogo uno alla volta: ciò causerebbe ritardi intollerabili. Nello stesso tempo è importante impedire agli scrittori di interferire tra loro, ed è necessario impedire ai lettori di leggere mentre ci sono delle operazioni di scrittura in corso, per evitare l'accesso ad informazioni non corrette.

È possibile considerare il problema del produttore/consumatore semplicemente come un caso speciale del problema dei lettori/scrittori con un solo scrittore (il produttore) e un solo lettore (il consumatore)? La risposta è no, perché il produttore non è solo uno scrittore, ma deve leggere i puntatori della coda per stabilire dove mettere il prossimo elemento, e deve stabilire se il buffer è pieno; analogamente il consumatore non è solo un lettore, perché deve modificare i puntatori della coda per riflettere il fatto che un elemento è stato rimosso dal buffer.

Ora saranno esaminate due soluzioni del problema.

Priorità ai lettori

La figura 5.28 è una soluzione che usa i semafori con un lettore e uno scrittore; la soluzione non cambia in caso di molti lettori e scrittori. Il processo scrittore è semplice: il semaforo *wsem* serve per garantire la mutua esclusione, e finché uno scrittore accede all'area condivisa, nessun altro scrittore o lettore vi può accedere. Anche il processo lettore usa il semaforo *wsem* per garantire la mutua esclusione, infatti nel caso di molti lettori bisogna che, quando nessun lettore sta leggendo, il primo lettore che tenta di leggere aspetti su *wsem*. Quando c'è già almeno un lettore che sta leggendo, i lettori successivi non devono aspettare prima di entrare. La variabile globale *numlettori* è usata per tenere traccia del numero di lettori, e il semaforo *x* è usato per assicurarsi che *numlettori* sia aggiornata correttamente.

Priorità agli scrittori

Nella soluzione precedente i lettori hanno la priorità: quando un lettore inizia ad accedere ai dati, i lettori possono mantenere il controllo dell'area dati finché c'è un lettore attivo, quindi gli scrittori rischiano un'attesa perenne.

La Figura 5.29 mostra una soluzione che impedisce a nuovi lettori di accedere ai dati se qualche scrittore ha dichiarato di voler effettuare una scrittura; sono stati aggiunti i seguenti semafori e variabili per gli scrittori:

- Un semaforo *rsem* che blocca i lettori quando c'è almeno uno scrittore che desidera accedere ai dati.
- Una variabile *numscrittori* che controlla il numero di scrittori per gestire *rsem*.
- Un semaforo *y* che controlla l'aggiornamento di *numscrittori*.

```
program lettori_scrittori;
var  numlettori: integer;
     x, wsem: semaforo (:= 1);
procedure lettore;
begin
   repeat
     wait (x);
     numlettori := numlettori + 1;
      if numlettori = 1 then wait (wsem);
     signal (x);
     LEGGI_UNITÀ;
     wait (x);
     numlettori := numlettori - 1;
      if numlettori = 0 then signal (wsem);
     signal (x)
   forever
end;
procedure scrittore;
begin
   repeat
     wait (wsem);
     SCRIVI_UNITÀ;
     signal (wsem)
   forever
end;
begin
   numlettori := 0;
   parbegin
     lettore;
     scrittore
   parend
end.
```

Figura 5.28 *Soluzione del problema dei lettori/scrittori con semafori: priorità ai lettori*

Per i lettori serve un ulteriore semaforo; non bisogna permettere che si formi una lunga coda su *rsem*, altrimenti gli scrittori non possono saltare la coda. Quindi ad un solo lettore è permesso di entrare nella coda di *rsem*, e tutti gli altri devono accodarsi su un semaforo *z* subito prima di aspettare su *rsem*. La Tabella 5.5 fa un riassunto delle possibilità.

Una soluzione alternativa, che dà priorità agli scrittori ed è implementata usando lo scambio di messaggi, è descritta nella Figura 5.30: in questo caso c'è un processo controllore che ha accesso ai dati condivisi; gli altri processi che desiderano accedere ai dati devono mandare un messaggio di richiesta al controllore, aspettare un messaggio di via libera da questo, e inviare il messaggio "ho finito" per indicare che hanno terminato l'accesso. Il controllore ha tre caselle, una per ogni tipo di messaggio che può ricevere.

Il processo controllore serve le richieste di scrittura prima di quelle di lettura per dare priorità agli scrittori; inoltre bisogna garantire la mutua esclusione, e per far ciò si usa una variabile

Tabella 5.5 *Stati possibili delle code di processi per il programma di Figura 5.29*

Solo lettori	• *wsem* settato • non ci sono code
Solo scrittori	• *wsem* e *rsem* sono settati • gli scrittori si accodano su *wsem*
Lettori e scrittori con primo un lettore	• *wsem* settato dal lettore • *rsem* settato dallo scrittore • tutti gli scrittori si accodano su *wsem* • un lettore si accoda su *rsem* • gli altri lettori si accodano su *z*
Lettori e scrittori con primo uno scrittore	• *wsem* settato dallo scrittore • *rsem* settato dallo scrittore • gli scrittori si accodano su *wsem* • un lettore si accoda su *rsem* • gli altri lettori si accodano su *z*

contatore, che è inizializzata ad un numero maggiore del numero massimo di lettori, in quest'esempio 100. L'azione del controllore si può riassumere in questo modo:

- Se *contatore* > 0 allora nessuno scrittore è in attesa, ed è possibile che vi siano dei lettori attivi, come che non ce ne siano. Bisogna prima ricevere tutti i messaggi "ho finito", in modo da eliminare tutti i lettori attivi; quindi soddisfare tutte le richieste di scrittura ed infine tutte le richieste di lettura.
- Se *contatore* = 0, allora la sola richiesta pendente è di scrittura; permette allo scrittore di procedere ed aspetta il messaggio di fine scrittura. Si può far procedere lo scrittore ed attendere il messaggio "ho finito".
- Se *contatore* < 0, allora uno scrittore ha fatto una richiesta e sta aspettando che i lettori rimanenti abbiano finito; quindi accetta solo i messaggi di tipo "ho finito".

5.8 Sommario

I temi centrali dei sistemi operativi moderni sono la multiprogrammazione, il multiprocessing e l'elaborazione distribuita; la concorrenza è fondamentale per questi sistemi e per la tecnologia di progettazione dei sistemi operativi. Quando molti processi sono eseguiti concorrentemente, sia veramente nel caso di un sistema multiprocessore, sia virtualmente nel caso di un sistema a singolo processore con multiprogrammazione, nascono i problemi di risoluzione dei conflitti e cooperazione.

I processi concorrenti possono interagire in vari modi; i processi che non vedono gli altri processi possono entrare in competizione per le risorse, come il tempo di elaborazione e l'accesso ai dispositivi di I/O. I processi possono vedere gli altri processi indirettamente, perché condi-

```
program lettori_scrittori;
var numlettori, numscrittori: integer;
   x, y, z, wsem, rsem: semaforo (:= 1);
procedure lettore;
begin
   repeat
      wait (z);
         wait (rsem);
            wait (x);
               numlettori := numlettori + 1;
               if numlettori = 1 then wait (wsem);
            signal (x);
         signal (rsem);
      signal (z);
      LEGGI_UNITÀ;
      wait (x);
         numlettori := numlettori - 1;
         if numlettori = 0 then signal (wsem);
      signal (x)
   forever
end;
procedure scrittore;
begin
   repeat
      wait (y);
         numscrittori := numscrittori + 1;
         if numscrittori = 1 then wait (rsem);
      signal (y);
      wait (wsem);
      SCRIVI_UNITÀ;
      signal (wsem);
      wait (y);
         numscrittori := numscrittori - 1;
         if numscrittori = 0 then signal (rsem);
      signal (y)
   forever
end;
begin
   numlettori, numscrittori := 0;
   parbegin
      lettore;
      scrittore
   parend
end.
```

Figura 5.29 *Soluzione del problema dei lettori/scrittori usando i semafori: priorità agli scrittori*

```
procedure lettorei;
var rmsg: messaggio;
begin
    repeat
        rmsg := i;
        send (richiesta_lettura, rmsg);
        receive (casellai, rmsg);
        LEGGI_UNITÀ;
        rmsg := i;
        send (finito, rmsg)
    forever
end;
procedure scrittorej;
var rmsg: messaggio;
begin
    repeat
        rmsg := j;
        send (richiesta_scrittura, rmsg);
        receive (casellaj, rmsg);
        SCRIVI_UNITÀ;
        rmsg := j;
        send (finito, rmsg)
    forever
end;
procedure controllore;
begin
    repeat
        if contatore > 0 do begin
            if non_vuoto (finito) then begin
                receive (finito, msg);
                contatore := contatore + 1
            end
            else if non_vuoto (richiesta_scrittura) then begin
                receive (richiesta_scrittura, msg);
                scrittore.id := msg.id;
                contatore := contatore - 100
            end
            else if non_vuoto (richiesta_lettura) then begin
                receive (richiesta_lettura, msg);
                contatore := contatore - 1;
                send (msg.id, "OK")
            end
        end;
        if contatore = 0 do begin
            send (scrittore.id, "OK");
            receive (finito, msg);
            contatore := 100
        end;
```

Figura 5.30 *Soluzione del problema dei lettori/scrittori usando lo scambio di messaggi*

```
        while contatore < 0 do begin
            receive (finito, msg);
            contatore := contatore + 1
        end
    forever
end.
```

Figura 5.30 *Continua*

vidono l'accesso ad un oggetto comune, come un blocco di memoria centrale o un file. Infine ci sono i processi che si vedono direttamente e cooperano tramite lo scambio di informazioni. I punti chiave da analizzare in queste interazioni sono la mutua esclusione e lo stallo.

La mutua esclusione è una condizione che si applica ad un insieme di processi concorrenti, di cui uno solo alla volta può avere accesso ad una data risorsa, o svolgere una funzione particolare. Le tecniche per la mutua esclusione si possono usare per risolvere conflitti, come la competizione per le risorse, e per sincronizzare i processi in modo da permetterne la cooperazione. Un esempio del secondo caso è il modello del produttore/consumatore, in cui un processo aggiunge dati ad un buffer, e uno o più processi estraggono tali dati dal buffer.

Gli algoritmi per la mutua esclusione sono molti e quello di Dekker è il più noto. L'approccio software solitamente porta ad avere un aumento dei tempi di calcolo, e il rischio di errori logici (bug) è alto; un secondo approccio per fornire la mutua esclusione prevede l'uso di istruzioni macchina speciali: ciò riduce il tempo di calcolo, ma è ancora inefficiente perché usa l'attesa attiva.

Un altro approccio alla mutua esclusione è quello di fornire dei meccanismi a livello di sistema operativo: le due tecniche più comuni sono i semafori e i messaggi. I semafori sono usati per mandare segnali fra processi, e forniscono un modo diretto per garantire la mutua esclusione. I messaggi sono utili per la mutua esclusione e per avere una vera forma di comunicazione fra processi.

5.9 Letture raccomandate

Nonostante l'età, [BRIN73], che ancora è stampato, è il testo sui sistemi operativi che contiene una delle trattazioni migliori della concorrenza, ed ha il vantaggio ulteriore di contenere molti problemi con soluzioni. [BEN82] presenta una discussione molto chiara e piacevole di concorrenza, mutua esclusione, semafori e altri argomenti correlati. Un trattamento più formale, esteso ai sistemi distribuiti, è contenuto in [BEN90]. [AXFO88] è un altro testo utile e di facile lettura, e contiene vari problemi con soluzioni. [RAYN86] è un insieme completo di algoritmi per la mutua esclusione, sia software (es. Dekker) sia hardware, nonché semafori e messaggi. [HOAR85] è un classico molto leggibile, che presenta un approccio formale alla definizione di processi sequenziali e concorrenza. [LAMP86] contiene una trattazione della mutua esclusione

lunga e formale. [RUDO90] è un utile aiuto per capire la concorrenza. [BACO93] tratta la concorrenza, in modo ben organizzato. [BUHR95] è un trattato esauriente sui monitor.

AXFO88 T. Axford. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1988.

BACO93 J. Bacon. *Concurrent Systems*. Reading, MA: Addison-Wesley, 1993.

BEN82 M. Ben-Ari. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.

BEN90 M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

BRIN73 P. Brinch-Hansen. *Operating System Principles*. Englewood Cliffs, NJ: Prentice Hall, 1973.

BUHR95 P. Buhr, M. Fortier. "Monitor Classification". *ACM Computing Surveys*, Marzo 1995.

HOAR85 C. Hoare. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.

LAMP86 L. Lamport. "The Mutual Exclusion Problem". *Journal of the ACM*, Aprile 1986.

RAYN86 M. Raynal. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.

RUDO90 B. Rudolph. "Self-Assessment Procedure XXI: Concurrency". *Communications of the ACM*, Maggio 1990.

5.10 Problemi

5.1 I processi e i thread forniscono un potente strumento di strutturazione per implementare programmi che in stile sequenziale sarebbero molto più complessi; la coroutine è un costrutto precedente, che è istruttivo analizzare. Lo scopo di questo problema è di introdurre le coroutine e confrontarle con i processi. Si consideri questo semplice problema tratto da [CONW63]:

Leggere schede con 80 colonne e stamparle in righe di 125 caratteri con le seguenti modifiche: dopo l'immagine di ogni scheda s'inserisce uno spazio bianco, e ad ogni coppia di asterischi adiacenti (**) su una scheda si sostituisce il carattere ≠.

a. Si sviluppi una soluzione del problema con un normale algoritmo sequenziale; si vedrà che il programma è difficile da scrivere: le interazioni fra i vari elementi del programma sono complesse, a causa della conversione da 80 a 125 colonne; inoltre, la lunghezza dell'immagine della scheda, dopo la conversione, varia a seconda del numero di coppie di asterischi presenti. Un modo per migliorare la chiarezza e minimizzare i possibili errori è di scrivere tre procedure separate. La prima legge l'immagine della scheda, aggiunge lo spazio alla fine e scrive i caratteri su un file temporaneo. Dopo aver letto tutte le schede, la seconda procedura legge il file temporaneo, effettua la sostituzione dei caratteri e scrive in un secondo file temporaneo. La terza procedura legge questo file e stampa le righe di 125 caratteri.

b. La soluzione sequenziale è poco attraente a causa del sovraccarico nelle operazioni di I/O e dei file temporanei. Conway ha proposto una nuova forma di programma, la coroutine, che permette di scrivere l'applicazione come formata da tre programmi, connessi da buffer di un carattere (Figura 5.31). In una **procedura** tradizionale c'è una relazione *master/slave* fra la procedura chiamante e quella chiamata: la procedura chiamante può eseguire una chiamata da un punto qualunque, mentre quella chiamata inizia dalla prima istruzione. La **coroutine** è caratterizzata da una relazione più simmetrica: ad ogni chiamata l'esecuzione inizia dall'ultimo punto attivo della procedura chiamata. Poiché non c'è distinzione fra procedura chiamante e procedura chiamata, non c'è un'istruzione di ritorno, ma le coroutine si passano il controllo tramite un comando di ripristino (*resume*). La prima volta che una coroutine è chiamata, è "ripristinata" alla prima istruzione; in seguito la coroutine è riattivata nel punto in cui ha eseguito l'ultimo comando di ripristino. Si noti che in un programma una sola coroutine alla volta può essere in esecuzione, e che i punti di transizione sono definiti esplicitamente nel codice, quindi non si tratta di concorrenza. Si spieghi il funzionamento del programma della Figura 5.31.

c. Il programma non tratta la condizione di terminazione; si supponga che la routine di I/O LEGGI_SCHEDA ritorni vero se ha scritto un'immagine di 80 carattere in *inbuf*, falso altrimenti. Si modifichi il programma per trattare questo caso; si noti che l'ultima riga stampata può contenere meno di 125 caratteri.

d. Si riscriva il programma usando tre processi e i semafori.

5.2 Si consideri un programma concorrente composto da due processi P e Q descritti in seguito; A, B, C, D ed E sono comandi atomici (indivisibili) arbitrari. Si supponga che il programma principale (non riportato) esegua i due processi in parallelo.

```
procedure P;
begin
    A;
    B;
    C;
end
```

```
procedure Q;
begin
    D;
    E;
end
```

Si elenchino tutte le possibili alternanze dei due processi (si mostrino le tracce di esecuzione a livello dei comandi atomici).

5.3 Si consideri il programma seguente:

```
const n = 50;
var cond: integer;
procedure totale;
var contatore: integer;
begin
    for contatore := 1 to n do cond := cond + 1
end;
```

```
var rs, sp: character;
     inbuf: array [1 . .80] of character;
     outbuf: array [1. .125] of character;
procedure leggi;
begin
    repeat
        LEGGI_SCHEDA (inbuf);
        for i=1 to 80 do
            begin
             rs := inbuf [i];
             RIPRISTINA squash
            end;
         rs := " ";
        RIPRISTINA squash
    forever
end;
procedure stampa;
begin
    repeat
        for j = 1 to 125
            begin
                outbuf [j] := sp;
                RIPRISTINA squash
            end;
        OUTPUT (outbuf)
     forever
end;
```

```
procedure squash;
begin
    repeat
        if rs π "≠" then
            begin
                sp := rs;
                RIPRISTINA stampa
            end
        else begin
            RIPRISTINA leggi;
            if rs = "*" then
                begin
                    sp := "≠";
                    RIPRISTINA stampa
                end
            else begin
                sp:= "*";
                RIPRISTINA stampa;
                sp := rs;
                RIPRISTINA stampa
            end
        end
        RIPRISTINA leggi
    forever
end.
```

Figura 5.31 *Una applicazione delle coroutine*

```
begin (*  programma principale *)
     cond := 0;
      parbegin
          totale; totale
     parend;
     write (cond)
end.
```

a. Si determini il valore finale massimo e minimo della variabile condivisa *cond* stampato da questo programma concorrente. Si supponga che la velocità relativa dei processi sia arbitraria e che si possa incrementare un valore solo dopo averlo caricato in un registro, con un'istruzione macchina separata.

b. Si supponga che un numero arbitrario di processi siano eseguiti in parallelo, con le ipotesi del punto (a). Qual è l'effetto di queste modifiche sui possibili valori finali di *cond*?

5.4 È vero che l'attesa attiva è sempre meno efficiente (in termini di tempo di processore) di una wait bloccante? Si giustifichi la risposta.

5.5 Si consideri il programma seguente:

```
var bloccato: array [0 ..1] of boolean;
    turno: 0 . . 1;
procedure P (id: integer);
begin
    repeat
        bloccato[id] := true;
        while turno ≠ id do
            begin
                while bloccato[1-id] do;
                turno := id
            end;
            < sezione critica >
            bloccato[id] := false;
            < resto del programma >
    until false
end;
begin
    bloccato[0] := false;
    bloccato[1] := false;
    turno := 0;
    parbegin
        P(0) ; P(1)
    parend
end.
```

Questa è una soluzione software del problema della mutua esclusione proposto in [HYMA66]. Si trovi un controesempio che dimostri che questa soluzione non è corretta. È interessante notare che questa soluzione è stata pubblicata in *Communications of the ACM*, ritenendola corretta.

5.6 Si dimostri la correttezza dell'algoritmo di Dekker.

a. Si dimostri la mutua esclusione. *Suggerimento*: si mostri che quando Pi entra nella sezione critica, l'espressione seguente è vera:

flag[*i*] **and** (**not** *flag*[*i*-1])

b. Si dimostri che un processo che chiede l'accesso alla propria sezione critica non deve aspettare indefinitamente. *Suggerimento*: si considerino i seguenti casi: (1) un solo processo tenta di entrare nella sezione critica; (2) entrambi i processi vogliono entrare, e (2a) *turno* = 0 e *flag*[0] = *false*, oppure (2b) *turno*=0 e *flag*[0]=*true*.

5.7 Si consideri l'algoritmo di Dekker, scritto per un numero arbitrario di processi cambiando il comando che è eseguito subito dopo la sezione critica da

turno:= 1 – *i* /* P0 mette turno *a* 1 *e* P1 mette turno a 0 */

a

turno : = (turno + 1) mod n /* n = numero di processi */

Si valuti l'algoritmo quando il numero di processi concorrenti è maggiore di due.

5.8 L'algoritmo di Peterson si può generalizzare per la mutua esclusione di *N* processi. Si usino due array globali, *q* e *turno*. Il valore iniziale degli *N* elementi di *q* e degli *N*-1 elementi di *turno* è 0. Ogni processo usa le variabili locali *j* e *k* come indici degli array. L'algoritmo del processo *i* è il seguente:

```
...
global integer arrays q [N], turno [N-1]

repeat
```

```
1  for j=1 to N-1 do {
2  q [i] = j;
3  turno [j] = i;
4  L:  for k=1 to i-1, i+1 to N
5      if ((q [k] ≥ j) and (turno [j] = i)) goto L;}
6  q [i] = N;
```

```
   sezione critica del processo i;
```

```
7  q [i] = 0;
```

```
   resto del processo i
until false
```

È utile pensare al valore della variabile locale *j* come lo "stadio" dell'algoritmo in cui il processo *i* è eseguito; l'array globale *q* indica lo stadio d'ogni processo. Quando un processo entra nella fase critica, passa allo stadio *N* (tramite il comando $q[i]=N$; in realtà questo comando è usato solo per semplificare la dimostrazione e non è necessario per la correttezza).

Se $q[x]>q[y]$ si dice che il processo *x* precede il processo *y*; sarebbe interessante dimostrare che l'algoritmo garantisce:

- La mutua esclusione.
- L'assenza di stallo.
- L'assenza di starvation.

Per farlo, si dimostrino i seguenti lemmi:

a. Lemma 1: un processo che precede tutti gli altri può procedere per almeno uno stadio. *Suggerimento*: si consideri che il processo *i* quando esegue la riga 4 troverà

$$j = q[i] > q[k] \qquad \text{per tutti i } k \neq i \tag{1}$$

b. Lemma 2: quando un processo passa dallo stadio j a $j+1$, si verificherà uno dei seguenti casi:

- Precede tutti gli altri processi.
- Non è l'unico nello stadio j.

Suggerimento: quando il processo i sta per incrementare $q[i]$, si controlli se l'equazione (1) è vera.

c. Lemma 3: se ci sono (almeno) due processi nello stadio j, c'è (almeno) un processo in ogni stadio k, $1 \leq k \leq j-1$.

Suggerimento: si dimostri per induzione su j.

d. Lemma 4: il numero massimo di processi che possono essere nello stadio j è $N-j+1$, $1 \leq j \leq N-1$.

Suggerimento: è un semplice calcolo di aritmetica.

e. Usando i Lemmi precedenti, si dimostri che l'algoritmo garantisce la mutua esclusione, l'assenza di stallo e l'assenza di starvation.

5.9 Un altro approccio software alla mutua esclusione è l'**algoritmo del panettiere** di Lamport [LAMP74], così chiamato perché si basa su quello che avviene nelle panetterie e in altri negozi: ogni cliente riceve un biglietto numerato all'ingresso e ognuno è servito a turno. L'algoritmo è il seguente:

```
var     scelta: array [0..n-1] of boolean;
  numero: array [0..n-1] of integer;
repeat
  scelta[i] := true;
  numero[i] := 1 + max(numero[0], numero[1] , .., numero[n-1]);
  scelta[i] := false;
  for j := 0 to n-1 do
       begin
        while scelta[j] do;
       while numero[j] ≠ 0 and (numero[j],j) < (numero[i],i) do;
       end;
  < sezione critica > ;
  numero[i] := 0;
  < resto del programma > ;
forever.
```

Gli array *scelta* e *numero* sono inizializzati rispettivamente a falso e 0. L'i-esimo elemento di ogni array può essere letto e scritto dal processo i, ma solo letto dagli altri processi. La notazione (a,b) < (c,d) sta per

$$(a < c) \text{ o } (a = c \text{ e } b < d)$$

a. Si descriva l'algoritmo a parole.

b. Si dimostri che l'algoritmo previene lo stallo.

c. Si mostri che la mutua esclusione è garantita.

5.10 Si dimostri che i seguenti approcci software alla mutua esclusione non dipendono dalla mutua esclusione elementare a livello di accesso alla memoria:

a. Algoritmo del panettiere.

b. Algoritmo di Peterson.

5.11 Quando si usa un'istruzione macchina speciale per garantire la mutua esclusione come nella Figura 5.11, non c'è controllo sul tempo massimo di attesa di un processo prima di poter accedere alla sezione critica. S'inventi un algoritmo che usi l'istruzione test-and-set, e che garantisca che ogni processo in attesa di entrare nella sezione critica lo possa fare entro al più n-1 turni, dove n è il numero di processi che possono richiedere l'accesso alla sezione critica, e si conta un turno quando un processo esce dalla sezione critica e un altro processo entra.

5.12 Si consideri la seguente definizione di semaforo:

```
wait(s):
     if s.contatore > 0
          then
               s.contatore := s.contatore - 1
     else begin
          aggiungi questo processo a s.coda;
          blocca
     end;
signal(s):
     if   c'è almeno un processo sospeso sul semaforo s
          then begin
               togli un processo P da s.coda;
               aggiungi P alla lista dei processi Ready
          end
     else
     s.contatore := s.contatore + 1
end;
```

Si confronti questa definizione con quella della Figura 5.8. Si noti una differenza: con la definizione precedente un semaforo non può mai avere un valore negativo. Ci sono differenze nell'uso delle due definizioni all'interno di programmi? Cioè, è possibile scambiare una definizione con l'altra senza modificare il significato del programma?

5.13 Dovrebbe essere possibile implementare i semafori generici usando semafori binari; si possono usare le operazioni waitB e signalB e due semafori binari, ritardo e mutex. Si consideri questo codice:

```
procedure Wait ( var s: semaforo );
```

```
begin
     WaitB(mutex);
     s := s - 1;
     if s < 0 then begin
          SignalB(mutex);
          WaitB(ritardo)
     end
     else
          SignalB (mutex)
end;
procedure Signal ( var s: semaforo );
begin
     WaitB(mutex);
     s := s + 1;
     if s <= 0 then SignalB(ritardo);
     SignalB(mutex)
end.
```

All'inizio ad s è assegnato il valore desiderato per il semaforo. Ogni Wait decrementa s, e ogni Signal lo incrementa. Il semaforo binario mutex, inizializzato a 1, assicura che ci sia mutua esclusione durante l'aggiornamento di s. Il semaforo binario ritardo, inizializzato a 0, è usato per sospendere i processi.
Nel programma precedente c'è un errore: si dica qual è e si proponga un rimedio. *Suggerimento*: è sufficiente spostare una riga di codice.

5.14 Il problema seguente è stato dato ad un esame: Jurassic Park è costituito da un museo di dinosauri e un parco safari. Ci sono m passeggeri e n vetture ad un posto. I passeggeri vagano per il museo per un po' di tempo, poi si mettono in coda per fare un giro nel parco safari. Quando una vettura è disponibile, carica un passeggero e gira per il parco per un tempo casuale. Se le n vetture sono tutte occupate per fare un giro, un passeggero che vuole fare un giro deve aspettare; se una vettura è libera ma non c'è nessun passeggero in attesa, la vettura aspetta. Si usino i semafori per sincronizzare gli m processi per i passeggeri e gli n processi per le vetture.
Il codice seguente è stato rinvenuto su un pezzo di carta sul pavimento dell'aula dopo l'esame. Si giudichi se è corretto, prescindendo dalla sintassi e dalle dichiarazioni di variabili mancanti.

```
resource Jurassic_Park()
sem vettura_disponibile := 0, riservata_vettura := 0;
sem vettura_occupata := 0, rilasciato_passeggero := 0;

process passeggero(i := 1 to num_passeggeri)
     do true -> pisolino(int(random(1000*durata_corsa)))
          P(vettura_disponibile);
          V(riservata_vettura);
          P(vettura_occupata);
          P(rilasciato_passeggero)
```

```
        od
    end passeggero
    process vettura(j := 1 to num_vetture)
        do true -> V(vettura_disponibile);
            P(riservata_vettura);
            V(vettura_occupata);
            pisolino(int(random(1000*durata_corsa)))
            V(rilasciato_passeggero)
        od
    end vettura
    end Jurassic_Park
```

5.15 Si consideri la soluzione del problema del produttore/consumatore con buffer infinito definita nella Figura 5.14. Si supponga che si verifichi la situazione comune in cui il produttore e il consumatore sono eseguiti circa alla stessa velocità. Lo scenario sarà il seguente:

Produttore: aggiungi; signal; produci; ...; aggiungi; signal; produci; ...
Consumatore: consuma; ...; togli; wait; consuma; ...; togli; wait; ...

Il produttore riesce sempre ad aggiungere un nuovo elemento al buffer e a segnalarlo al consumatore mentre questo consuma l'elemento precedente. Il produttore aggiunge sempre elementi al buffer vuoto, e il consumatore prende sempre il solo elemento contenuto nel buffer. Anche se il consumatore non sarà mai bloccato su un semaforo, effettuerà un gran numero di operazioni sui semafori, che provocano un sovraccarico notevole.
Si scriva un nuovo programma che in queste circostanze sia più efficiente.
Suggerimento: si permetta che n assuma il valore -1, ad indicare che non solo il buffer è vuoto, ma che il consumatore ne è al corrente e sarà bloccato finché il produttore non aggiungerà un elemento. La soluzione non richiede l'uso della variabile locale m nella Figura 5.14.

5.16 Si consideri la Figura 5.17. Quale dei seguenti scambi modificherebbe il significato del programma?

a. wait(e); wait(s)

b. signal(s); signal(n)

c. wait(n); wait(s)

d. signal(s); signal(e)

5.17 Nella discussione del problema del produttore/consumatore con buffer limitato, si noti che il buffer poteva contenere al più n-1 elementi.

a. Qual è la ragione?

b. Si modifichi l'algoritmo per superare questa limitazione.

5.18 Si risponda a queste domande a proposito del barbiere equo (Figura 5.21).

a. È vero che il programma stabilisce che il barbiere che finisce di tagliare i capelli ad un cliente è quello che si fa pagare da quel cliente?

b. È vero che ogni barbiere usa sempre la stessa sedia?

5.19 La soluzione proposta nella Figura 5.21 ha ancora alcuni problemi; si modifichi il programma per correggere i problemi seguenti.

a. Può succedere che il cassiere accetti il pagamento di un cliente e ne liberi un altro, se due o più clienti aspettano per pagare. Fortunatamente quando un cliente paga non ha modo di riavere i soldi indietro, così alla fine il registratore di cassa contiene la giusta quantità di soldi, in ogni modo è meglio liberare il cliente giusto dopo che questo ha pagato.

b. Il semaforo *lascia_sedia_b* dovrebbe impedire accessi multipli ad una sedia, ma sfortunatamente questo semaforo non garantisce sempre che questo avvenga. Ad esempio, si supponga che tutti e tre i barbieri abbiano finito il taglio e siano bloccati su *wait(lascia_sedia_b)*. Due clienti sono stati interrotti subito prima di lasciare la sedia, e il terzo lascia la sedia ed esegue *signal(lascia_sedia_b)*. Quale barbiere è liberato? Poiché la coda *lascia_sedia_b* è first-in-first-out, sarà liberato il barbiere che è stato bloccato per primo; non è detto che sia proprio quello che stava tagliando i capelli al cliente che ha eseguito la signal. Se non lo fosse, un nuovo cliente si sederà in braccio ad un cliente che stava per alzarsi.

c. Il programma richiede che i clienti si siedano sul divano anche se c'è una sedia libera. Questo è un piccolo problema, e per risolverlo è necessario complicare ulteriormente il codice già complesso, in ogni modo si provi a farlo.

5.20 Questo problema mostra l'uso dei semafori per coordinare tre tipi di processi[4]. Babbo Natale dorme nel suo negozio al Polo Nord e può essere svegliato solo se (1) tutte e nove le renne tornano dalla vacanza nel Sud del Pacifico, o (2) un folletto ha problemi nella fabbricazione di un giocattolo. Per permettere a Babbo Natale di dormire, i folletti lo svegliano solo quando tre di loro hanno problemi. Mentre Babbo Natale risolve i problemi di tre folletti, tutti gli altri folletti che hanno bisogno d'aiuto devono aspettare. Se Babbo Natale si sveglia trovando tre folletti che lo aspettano davanti alla porta insieme con l'ultima renna di ritorno dai tropici, decide che i folletti possono aspettare fino a Natale, perché è più importante preparare la slitta (si suppone che le renne vogliano restare ai tropici fino all'ultimo momento). L'ultima renna va da Babbo Natale, mentre le altre aspettano in un riparo al caldo, in attesa di essere legate alla slitta. Si risolva questo problema mediante l'uso di semafori.

5.21 Si dimostri che lo scambio di messaggi e i semafori hanno le stesse funzionalità:

a. Implementando lo scambio di messaggi tramite semafori. *Suggerimento*: si utilizzi un buffer condiviso per le caselle postali, implementate come array di messaggi.

b. Si implementi un semaforo usando lo scambio di messaggi. *Suggerimento*: si utilizzi un processo indipendente per la sincronizzazione.

---

[4] Sono grato a John Trono del St. Michael's College nel Vermont per aver fornito questo problema.

C A P I T O L O 6

CONCORRENZA: STALLO E STARVATION

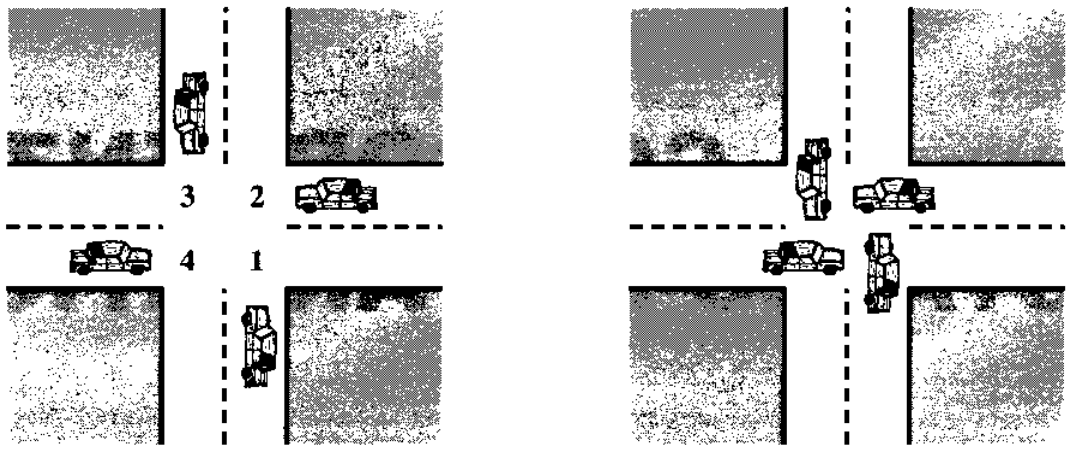
Questo capitolo continua lo studio della concorrenza analizzando due problemi che si presentano in ogni applicazione di calcolo concorrente: stallo e starvation. Si inizia con una discussione sui principi dello stallo e il problema correlato della starvation. In seguito si analizzano i tre approcci più comuni alla gestione dello stallo: prevenzione, rilevamento ed esclusione. Poi si studia un classico problema che illustra i problemi di sincronizzazione e stallo: il problema dei filosofi a tavola.

Come nel Capitolo 5, la discussione si limita a considerazioni su stallo e starvation in un singolo sistema. Lo stallo distribuito sarà studiato nel Capitolo 14.

6.1 Principi dello stallo

Lo stallo si può definire come il *blocco permanente* di un insieme di processi che competono per le risorse di sistema o comunicano fra loro. A differenza degli altri problemi di gestione dei processi, non c'è una soluzione efficiente per il caso generale.

Tutti gli stalli sono dovuti a un conflitto di richieste di risorse da parte di due o più processi. Un esempio comune è lo stallo dovuto al traffico; la Figura 6.1a mostra una situazione in cui quattro macchine arrivano nello stesso istante ad un incrocio a 4 vie. Le risorse sono i quattro quadranti dell'incrocio; in particolare, se tutte e quattro le macchine vogliono proseguire diritto dopo l'incrocio, le richieste di risorse sono le seguenti:



(a) Stallo possibile (b) Stallo

Figura 6.1 *Illustrazione di uno stallo*

- La macchina che va a nord vuole i quadranti 1 e 2.
- La macchina che va a ovest vuole i quadranti 2 e 3.
- La macchina che va a sud vuole i quadranti 3 e 4.
- La macchina che va a est vuole i quadranti 4 e 1.

Il codice della strada impone che una macchina ferma ad un incrocio a quattro vie deve dare la precedenza alle macchine provenienti da destra; questa regola funziona se ci sono solo due o tre macchine ferme all'incrocio: ad esempio, se ci sono solo le macchine dirette a nord e a ovest, quella diretta a nord deve aspettare e l'altra può procedere. Comunque, se tutte e quattro le macchine arrivano nello stesso istante, nessuna potrà oltrepassare l'incrocio, causando uno stallo. Se le quattro macchine ignorano le regole e proseguono (con cautela!), ognuna occuperà una risorsa (quadrante), ma non potrà procedere perché la seconda risorsa necessaria sarà già occupata da un'altra macchina. Anche in questo caso si ha stallo.

Si consideri lo stallo di processi che richiedono risorse del computer: la Figura 6.2 illustra il progresso di due processi che competono per due risorse. Ogni processo ha bisogno dell'uso esclusivo di entrambe le risorse per un certo periodo di tempo. Il processo P ha la seguente forma

```
Processo P
...
Prendi A
...
Prendi B
```

```
...
Rilascia A
...
Rilascia B
...
```

e il processo Q ha la forma

```
Processo Q
...
Prendi B
...
Prendi A
...
Rilascia B
...
Rilascia A
...
```

Nella Figura 6.2 l'asse *x* rappresenta il progresso dell'esecuzione di P e l'asse *y* quello di Q. L'unione dei progressi dei due processi è rappresentata da un cammino, che parte dall'origine e prosegue verso nord o verso est. Nel caso di un sistema monoprocessore, un solo processo alla volta può essere in esecuzione, e il cammino consiste di un'alternanza di cammini orizzontali e verticali, dove un segmento orizzontale rappresenta un periodo in cui P è in esecuzione e Q aspetta, e un segmento verticale viceversa.

La Figura 6.2 mostra sei cammini di esecuzione diversi, che si possono riassumere così:

1. Q ottiene B e poi A, poi rilascia B e A. Quando P è riattivato, può avere entrambe le risorse.
2. Q ottiene B e poi A. P viene eseguito e si blocca sulla richiesta di A. Q rilascia B e A. Quando P è riattivato, può avere entrambe le risorse.
3. Q ottiene B e P ottiene A. Lo stallo è inevitabile, perché Q verrà bloccato su A e P su B.
4. P ottiene A e poi Q ottiene B. Lo stallo è inevitabile, perché Q verrà bloccato su A e P su B.
5. P ottiene A e poi B. Q viene bloccato sulla richiesta di B. P rilascia A e B. Quando Q viene riattivato, può avere entrambe le risorse.
6. P ottiene A e poi B, e poi rilascia A e B. Quando Q viene riattivato, può avere entrambe le risorse.

La possibilità di avere stallo dipende sia dalla dinamica dell'esecuzione sia dai dettagli dell'applicazione. Ad esempio, si supponga che P non abbia bisogno di entrambe le risorse nello stesso tempo, e abbia la forma seguente:

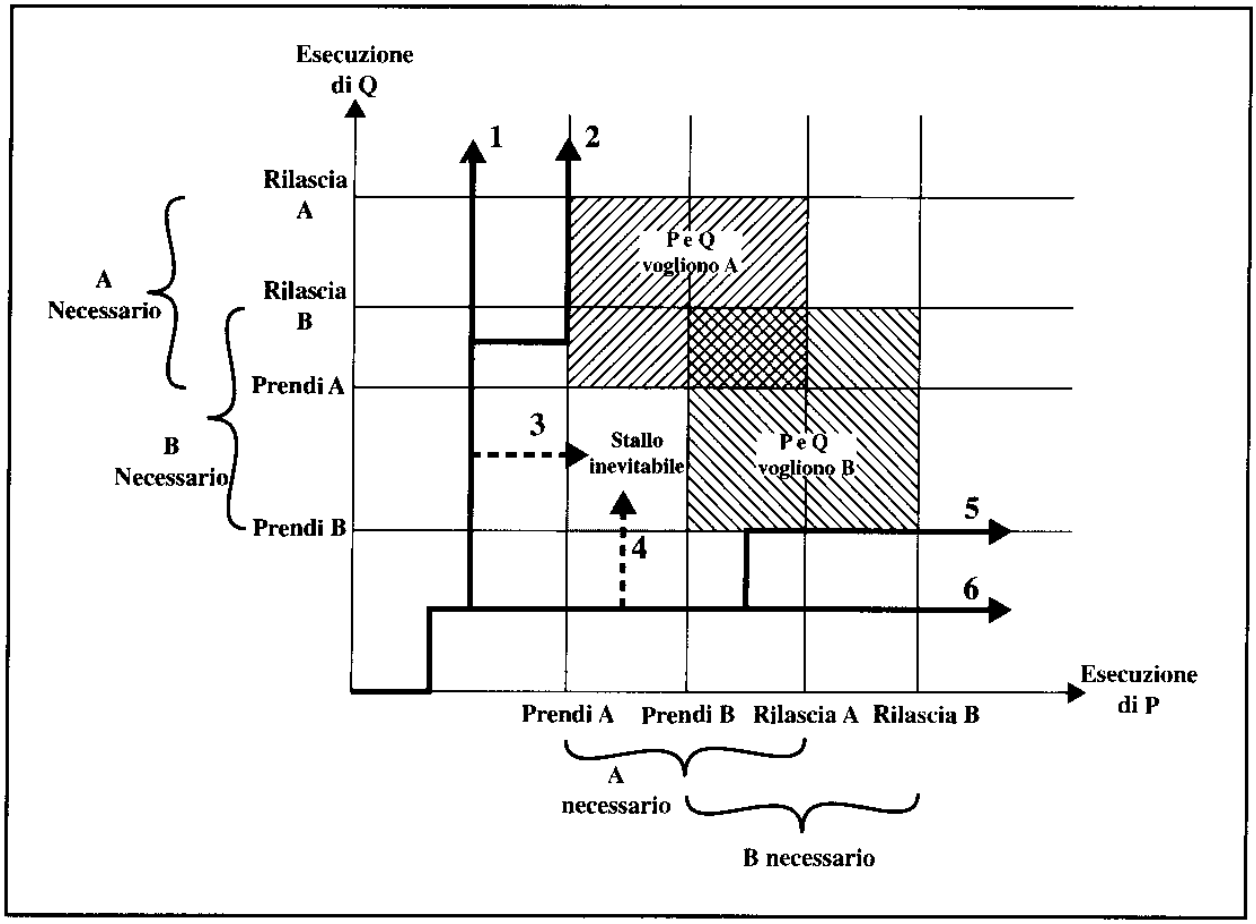


Figura 6.2 *Esempio di stallo [BACO93]*

```
Processo P
...
Prendi A
...
Rilascia A
...
Prendi B
...
Rilascia B
...
```

Questa situazione si riflette nella Figura 6.3; si noti che non si può avere stallo, qualunque sia la velocità relativa dei due processi.

Risorse riutilizzabili

Si distinguono due categorie di risorse: riutilizzabili e non riutilizzabili. Una risorsa riutilizzabile può essere usata da un processo alla volta e non viene distrutta dopo l'uso; i processi

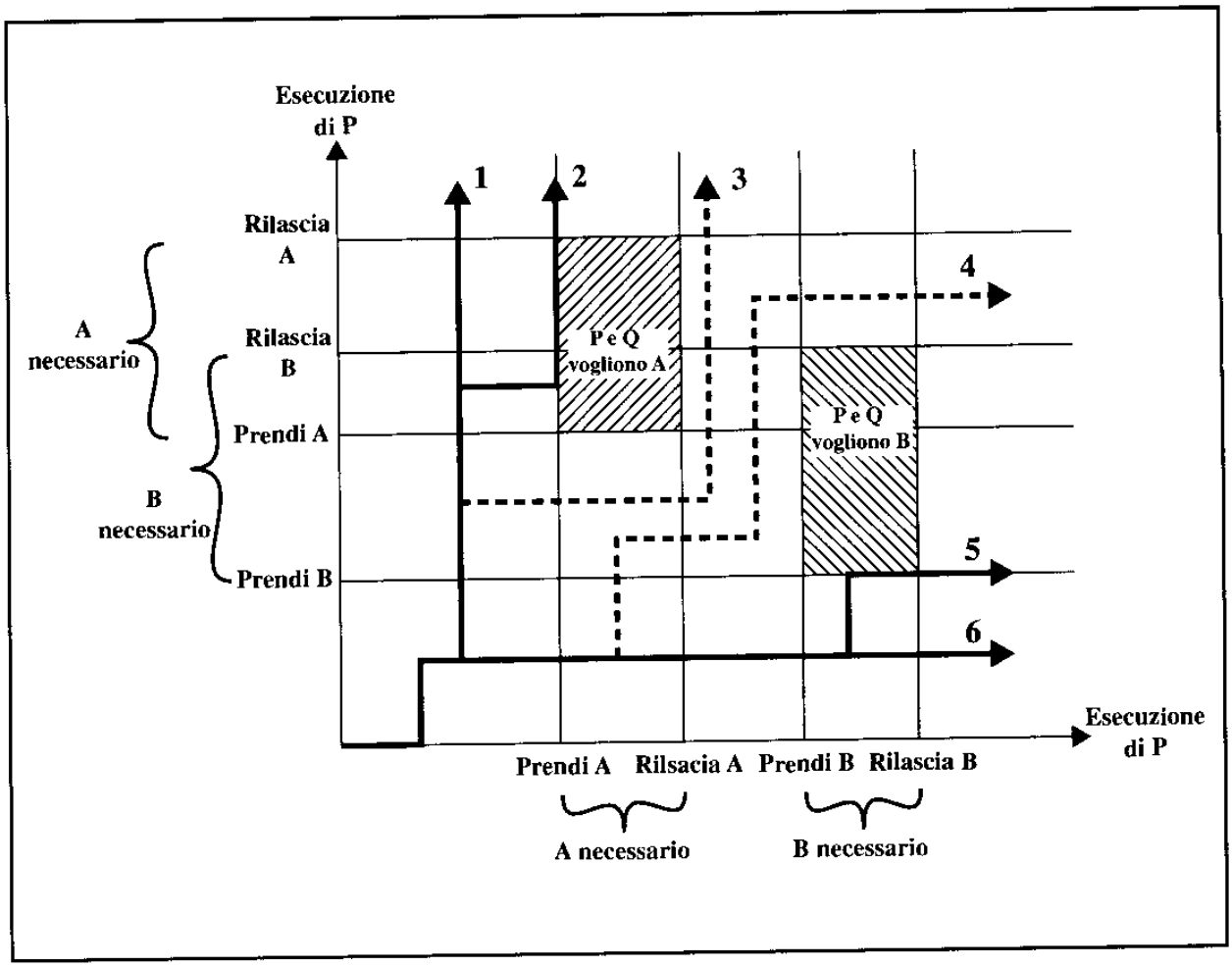


Figura 6.3 *Esempio di assenza di stallo [BACO93]*

prendono possesso di queste risorse, e poi le rilasciano per permettere agli altri processi di utilizzarle. Esempi di risorse riutilizzabili sono processori, canali di I/O, memoria centrale e secondaria, dispositivi, e strutture dati come file, basi di dati e semafori.

Come esempio di stallo con risorse riutilizzabili, si considerino due processi in competizione per l'accesso esclusivo ad un file D e ad un registratore di nastri T. I programmi effettuano le operazioni mostrate nella Figura 6.4. Si ha stallo se il sistema di multiprogrammazione alterna le esecuzioni dei due processi nel modo seguente:

```
p0p1q0q1p2q2
```

A prima vista questo sembra un errore di programmazione più che un problema di progettazione del sistema operativo, comunque si è visto che la progettazione di programmi concorrenti è difficile. Questi stalli capitano, e spesso la causa è nascosta dentro la complessa logica del programma, quindi sono difficili da rilevare. Una strategia per gestire gli stalli di questo tipo è di imporre, a livello di progettazione, dei vincoli sull'ordine delle richieste di risorse.

Un altro esempio di risorsa riutilizzabile che può provocare stallo è la memoria centrale. Si supponga che lo spazio disponibile sia di 200 KB, e che ci sia questa sequenza di richieste:

```
P1
...
Richiedi 80 kbyte;
...
Richiedi 60 kbyte;
```

```
P2
...
Richiedi 70 kbyte;
...
Richiedi 80 kbyte;
```

Lo stallo avviene se entrambi i processi effettuano la seconda richiesta. Se non è possibile conoscere in anticipo la quantità di memoria che sarà richiesta, gestire questo tipo di stallo tramite vincoli di sistema è difficile. Il modo migliore di gestire il problema è di eliminare questa possibilità mediante l'uso della memoria virtuale, argomento trattato nel Capitolo 8.

Risorse non riutilizzabili

Una risorsa è detta non riutilizzabile se viene distrutta (consumata) dopo essere stata creata (prodotta). Tipicamente non c'è limite al numero di risorse non riutilizzabili di un tipo particolare; un processo produttore, se non è bloccato, può fornire un numero illimitato di queste risorse. Quando un processo acquisisce una risorsa, questa cessa di esistere. Esempi di tali risorse sono gli interrupt, i segnali, i messaggi e i dati contenuti nei buffer di I/O.

Per vedere un esempio di stallo con risorse non riutilizzabili si consideri la seguente coppia di processi:

```
P1
...
Receive(P2);
...
Send(P2);
```

```
P2
...
Receive(P1);
...
Send(P1);
```

Processo P

Passo	Azione
p_0	Richiedi (D)
p_1	Blocca (D)
p_2	Richiedi (T)
p_3	Blocca (T)
p_4	Esegui funzione
p_5	Rilascia (D)
p_6	Rilascia (T)

Processo Q

Passo	Azione
q_0	Richiedi (T)
q_1	Blocca (T)
q_2	Richiedi (D)
q_3	Blocca (D)
q_4	Esegui funzione
q_5	Rilascia (T)
q_6	Rilascia (D)

Figura 6.4 *Esempio di due processi in competizione per risorse riutilizzabili*

Tabella 6.1 *Riassunto degli approcci di prevenzione, rilevamento ed esclusione dello stallo nei sistemi operativi [ISLO80]*

Principio	Politica di allocazione delle risorse	Schemi possibili	Vantaggi principali	Svantaggi principali
Prevenzione	Conservativa; risorse sottoutilizzate.	Richiedere tutte le risorse contemporaneamente	• Funziona bene per i processi che effettuano lunghe sequenze di operazioni • Prerilascio non necessario	• Inefficiente • Ritarda l'avvio dei processi
		Prerilascio	• Conveniente se si usano risorse il cui stato può essere salvato e ripristinato facilmente	• Il prerilascio avviene più spesso del necessario • Soggetto a riattivazioni cicliche
		Ordinamento delle risorse	• Si può realizzare con controlli a tempo di compilazione • Non richiede tempo di esecuzione perchè il problema viene risolto a livello di progettazione del sistema	• Il prerilascio viene effettuato in modo non efficace • Non permette richieste di risorse incrementali
Rilevamento	Molto liberale; le richieste vengono soddisfatte e poi si controlla la presenza di stallo.	Chiamato periodicamente per controllare la presenza di stallo.	• Non ritarda mai l'avvio di un processo • Facilita la gestione online	• Svantaggi intrinseci del prerilascio
Esclusione	Intermedia fra prevenzione e rilevamento	Algoritmo per trovare almeno un cammino sicuro	• Il prerilascio non è necessario	• Bisogna conoscere a priori le future richieste di risorse • I processi possono essere bloccati per lunghi periodi

Se la *Receive* è bloccante, si ha stallo; ancora una volta, la causa è un errore di progettazione. Tali errori possono essere subdoli e difficili da scovare; inoltre è possibile che lo stallo avvenga solo dopo una rara combinazione di eventi, quindi un programma potrebbe funzionare per parecchio tempo, anche anni, prima di evidenziare un problema.

Non esiste un'unica strategia efficace per trattare tutti i casi di stallo. La Tabella 6.1 riassume gli elementi chiave degli approcci più importanti che sono stati sviluppati: prevenzione, rilevamento ed esclusione. Ciascuno sarà esaminato a turno.

Condizioni per lo stallo

Affinché lo stallo sia possibile, si devono verificare tre condizioni:

1. **Mutua esclusione.** Un solo processo alla volta può usare una data risorsa.
2. **Possesso e attesa.** Un processo può mantenere il possesso delle risorse allocate mentre aspetta di avere altre risorse.
3. **Assenza di prerilascio.** Se un processo possiede una risorsa, non può essere forzato a rilasciarla.

In molti casi queste condizioni sono desiderabili; ad esempio la mutua esclusione è necessaria per assicurare la coerenza dei risultati e l'integrità di una base di dati. Analogamente il prerilascio non si può applicare in modo arbitrario e, in special modo quando le risorse in gioco sono dati, deve essere associato ad un meccanismo di ripristino, che riporta un processo e le sue risorse ad uno stato precedente, da cui il processo può riprendere l'esecuzione.

Si può avere stallo in presenza di queste tre condizioni, ma si potrebbe anche non averlo: lo stallo avviene realmente quando una quarta condizione è soddisfatta.

4. **Attesa circolare.** C'è una catena chiusa di processi, tale che ogni processo possiede almeno una risorsa richiesta dal processo successivo nella catena (es. Figura 6.5).

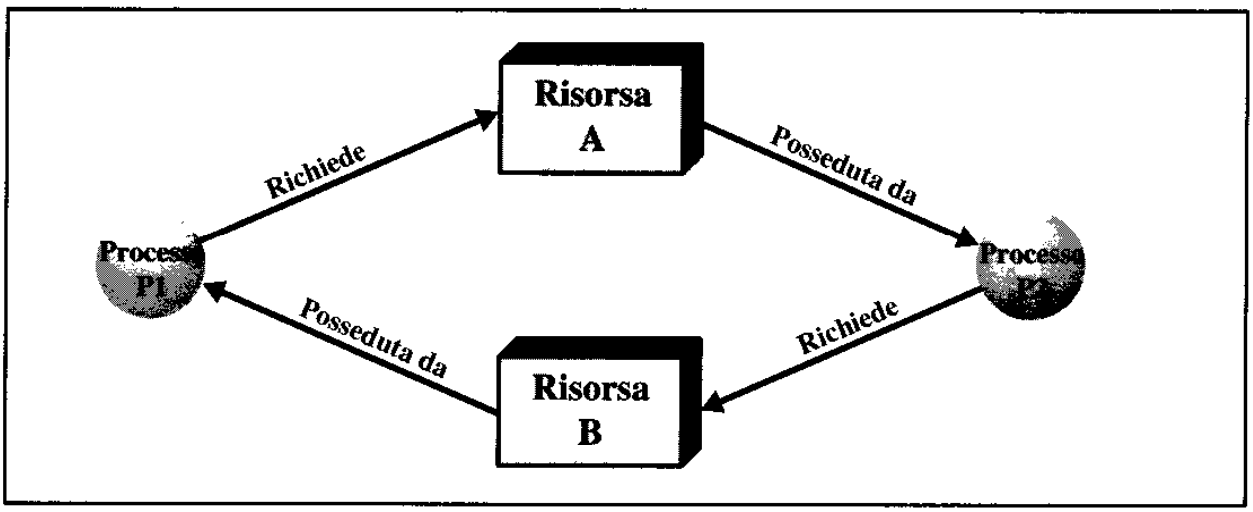


Figura 6.5 *Attesa circolare*

Le prime tre condizioni sono necessarie ma non sufficienti per avere stallo, e la quarta condizione è in realtà una conseguenza potenziale delle prime tre: in presenza delle prime tre condizioni, si può avere una sequenza di eventi che portano ad una situazione di attesa circolare non risolvibile, e questa coincide con la definizione di stallo. L'attesa circolare non è risolvibile quando le prime tre condizioni sono soddisfatte, quindi le quattro condizioni insieme formano una condizione necessaria e sufficiente per lo stallo[1].

6.2 Prevenzione dello stallo

La strategia di prevenzione dello stallo, semplificando, consiste nel progettare un sistema in modo che la possibilità di avere stallo sia esclusa a priori. Si distinguono due classi di prevenzione: un metodo indiretto è quello che previene il verificarsi di una delle tre condizioni necessarie (dalla 1 alla 3), mentre un metodo diretto previene il verificarsi di un'attesa circolare (condizione 4). Si esaminano ora le tecniche correlate a ciascuna condizione.

Mutua esclusione

In generale la prima delle quattro condizioni potrà verificarsi; se l'accesso ad una risorsa richiede mutua esclusione, allora questa deve essere supportata dal sistema operativo. Alcune risorse, come i file, permettono accessi multipli in lettura, ma un solo accesso esclusivo in scrittura; anche in questo caso si può avere stallo se più processi richiedono il permesso di effettuare una scrittura.

Possesso e attesa

La condizione di possesso-e-attesa si può prevenire imponendo che ogni processo richieda tutte le risorse all'inizio e che venga bloccato finché tutte le richieste non possono essere soddisfatte contemporaneamente. Questo approccio è inefficiente per due ragioni: la prima è che un processo è forzato ad aspettare che tutte le risorse siano disponibili, mentre in realtà potrebbe procedere utilizzandone solo una parte. La seconda ragione è che le risorse assegnate ad un processo possono rimanere inutilizzate per lungo tempo, rimanendo inaccessibili agli altri processi.

C'è anche il problema pratico creato dall'uso della programmazione modulare o dalla struttura multithread di un'applicazione: questa dovrebbe conoscere a priori quali sono le risorse che verranno richieste a tutti i livelli e in tutti i moduli per effettuare un'unica richiesta.

---

[1] Praticamente tutti i libri di testo elencano queste condizioni semplicemente come le condizioni necessarie per lo stallo, ma una tale presentazione nasconde alcuni problemi subdoli [SHUB90]. Il punto 4, la condizione di attesa circolare, è fondamentalmente diversa dalle altre. Le prime tre sono condizioni di progetto, mentre la quarta è un evento che si può verificare, e dipende dalla particolare sequenza di richieste e rilasci di risorse da parte dei processi coinvolti. Mettere la quarta condizione sullo stesso piano delle prime tre porta ad una distinzione inadeguata fra prevenzione ed esclusione.

Assenza di prerilascio

Questa condizione si può prevenire in diversi modi: ad esempio se ad un processo in possesso di alcune risorse vengono rifiutate ulteriori richieste, quel processo deve rilasciare le risorse necessarie e, se necessario, richiederle nuovamente, insieme con le nuove risorse. In alternativa, se un processo richiede una risorsa che al momento è posseduta da un altro processo, il sistema operativo può interrompere il secondo processo e forzarlo a rilasciare la risorsa; in questo caso si può prevenire lo stallo solo se ogni processo ha una priorità diversa dagli altri.

Questo approccio è pratico solo se applicato a risorse il cui stato può essere facilmente salvato e ripristinato in seguito, come nel caso di un processore.

Attesa circolare

La condizione di attesa circolare si può prevenire definendo un ordine lineare fra i tipi di risorsa; se un processo possiede delle risorse di tipo R, in seguito può richiedere solo risorse il cui tipo segue R nell'ordinamento.

Per capire il funzionamento di questa strategia, si associ un indice ad ogni tipo di risorsa. La risorsa R_i precede R_j nell'ordine se $i < j$. Si supponga che due processi, A e B, formino uno stallo perché A possiede R_i e richiede R_j, mentre B possiede R_j e richiede R_i: tale condizione è impossibile perché implica $i < j$ e $j < i$.

Come per la prevenzione di possesso-e-attesa, la prevenzione dell'attesa circolare può essere inefficiente, rallentando i processi e impedendo senza ragione l'uso di risorse da parte di altri processi.

6.3 Esclusione dello stallo

L'esclusione dello stallo è un approccio che differisce in modo sottile dalla prevenzione dello stallo[2]. Nel caso della prevenzione dello stallo, si impongono dei vincoli sulla richiesta di risorse per prevenire almeno una delle quattro condizioni per lo stallo; ciò può avvenire indirettamente, prevenendo una delle tre condizioni di progetto (mutua esclusione, possesso-e-attesa, assenza di prerilascio), o direttamente, prevenendo l'attesa circolare. Questo provoca un uso inefficiente delle risorse e l'esecuzione inefficiente dei processi. L'esclusione dello stallo, invece, permette le tre condizioni necessarie, ma effettua delle scelte oculate, in modo da assicurarsi che le situazioni di stallo non vengano mai raggiunte; perciò l'esclusione permette più concorrenza rispetto alla prevenzione. Con l'esclusione, si decide dinamicamente se una richiesta di risorsa può portare al verificarsi di uno stallo, se verrà soddisfatta, quindi si richiede una conoscenza delle richieste future dei processi.

In questa sezione verranno descritti due approcci all'esclusione dello stallo:

---

[2] Il termine *esclusione* può creare confusione, infatti le strategie discusse in questa sezione possono essere considerate esempi di prevenzione dello stallo, perché in realtà prevengono il verificarsi di stalli.

- Un processo non viene avviato se le sue richieste possono provocare stallo.
- Non si concedono ulteriori risorse ad un processo se queste possono causare uno stallo.

Rifiuto del permesso di esecuzione

In un sistema di n processi e m tipi di risorse, si considerino i seguenti vettori e matrici:

Risorse = $(R_1, R_2, \ldots, R_m)$ — numero totale di risorse nel sistema per ciascun tipo

Disponibili = $(V_1, V_2, \ldots, V_m)$ — numero totale di risorse non assegnate ad un processo

$$\text{Richieste} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$$ richiesta di risorse per ogni processo

$$\text{Assegnazione} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$$ assegnazione corrente

La matrice Richieste indica il numero massimo di richieste di ogni processo per ogni risorsa, con una riga dedicata ad ogni processo, cioè C_{ij} = richieste della risorsa j da parte del processo i. Queste informazioni devono essere dichiarate all'inizio affinché l'esclusione funzioni. In maniera simile A_{ij} = assegnazione della risorsa j al processo i. Si può vedere che valgono le relazioni seguenti:

1. $R_i = V_i + \sum_{k=1}^{n} A_{ki}$ per tutti gli i: Ogni risorsa è assegnata o è disponibile
2. $C_{ki} \leq R_i$ per ogni k, i: nessun processo può richiedere più risorse del numero totale nel sistema
3. $A_{ki} \leq C_{ki}$ per ogni k, i: nessun processo può possedere più risorse di un tipo di quante ne ha richiesto all'inizio

Con queste definizioni è possibile descrivere una strategia di esclusione dello stallo che non permette ad un processo di iniziare l'esecuzione se le sue richieste potrebbero portare ad uno stallo.

Un nuovo processo P_{n+1} può partire solo se

$$R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki} \quad \text{per ogni } i$$

Cioè un processo può partire solo se è possibile soddisfare la somma delle sue richieste e di quelle dei processi correnti. Questa strategia non è certo ottimale, perché considera il caso peggiore: quello in cui tutti i processi fanno tutte le richieste nello stesso momento.

Rifiuto di allocare le risorse

La strategia di rifiutare l'allocazione di nuove risorse, nota come l'**algoritmo del banchiere**[3], fu proposta per la prima volta in [DIJK65]. Si inizia definendo le nozioni di stato e stato sicuro. Si consideri un sistema con un numero fissato di processi e risorse: in ogni istante un processo avrà zero o più risorse assegnate. Lo **stato** di un sistema è semplicemente l'assegnazione corrente di risorse ai processi, quindi consiste di due vettori, Risorse e Disponibili, e due matrici, Richieste e Assegnazione, come visto prima. Uno **stato sicuro** è uno stato in cui c'è almeno una sequenza che permette a tutti i processi di completare l'esecuzione senza avere stallo, mentre, naturalmente, uno **stato non sicuro** è uno stato che non è sicuro.

L'esempio seguente illustra questi concetti; la Figura 6.6a mostra lo stato di un sistema che contiene quattro processi e tre risorse. Il numero totale di risorse di tipo R1, R2 e R3, è rispettivamente di 9, 3 e 6 unità. Nello stato corrente le assegnazioni ai processi sono tali da lasciare libera un'unità della risorsa 2 e una della risorsa 3. La domanda è: questo stato è sicuro? Per rispondere alla domanda, si consideri una domanda intermedia: è possibile per tutti i processi completare l'esecuzione con le risorse disponibili? In altre parole, è possibile, con le risorse disponibili, coprire la differenza fra l'assegnazione attuale e le richieste massime dei processi? È evidente che questo non è possibile per P1, che ha una sola unità di R1 e richiede due ulteriori unità di R1, due unità di R2 e due di R3. In ogni modo, assegnando un'unità di R3 al processo P2, questo otterrebbe il numero massimo di risorse richieste e potrebbe completare l'esecuzione. Si supponga che ciò si verifichi: quando P2 completa l'esecuzione, le sue risorse diventano nuovamente disponibili, e lo stato risultante è mostrato nella Figura 6.6b. A questo punto la domanda è se tutti i processi rimanenti possono completare l'esecuzione; in questo caso la risposta è affermativa. Nel caso di P1, è possibile assegnargli tutte le risorse che ha richiesto, aspettare che completi l'esecuzione e rendere disponibili tutte le risorse che possedeva; lo stato risultante è quello della Figura 6.6c. In seguito si può eseguire P3, ottenendo lo stato della Figura 6.6d; per concludere, si esegue P4. A questo punto tutti i processi hanno completato l'esecuzione, quindi lo stato descritto nella Figura 6.6a è uno stato sicuro.

Questi concetti suggeriscono una strategia di esclusione dello stallo, che assicura che il sistema di processi e risorse sia sempre in uno stato sicuro. Quando un processo richiede un insieme di risorse, si suppone che queste siano concesse, si aggiorna lo stato del sistema e si determina se il risultato è uno stato sicuro. In caso positivo si accetta la richiesta, altrimenti si blocca il processo finché la richiesta non può essere soddisfatta.

Si consideri lo stato definito dalla matrice della Figura 6.7a. Si supponga che P2 richieda un'unità di R1 e una di R3; se la richiesta è soddisfatta, allora lo stato risultante è quello della Figura 6.6a. Come si è visto prima, questo è uno stato sicuro, quindi è possibile soddisfare la

---

[3] Dijkstra usò questo nome per l'analogia di questo problema con quello che accade in una banca, dove i clienti che chiedono un prestito corrispondono ai processi, e il denaro corrisponde alle risorse. Nella terminologia delle banche: la banca ha una riserva limitata di denaro da prestare, e una lista di clienti, ciascuno con un conto debiti. Un cliente può scegliere di prelevare dal conto debiti un po' alla volta, e non c'è garanzia che il cliente sia in grado di restituire il denaro prima di aver prelevato il massimo consentito dal suo conto. Il banchiere può rifiutare un prestito ad un cliente se c'è il rischio che la banca non abbia fondi sufficienti per permettere ai clienti di prelevare il denaro, prima che essi ripaghino la banca.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
9	3	6

Vettore delle risorse

R1	R2	R3
0	1	1

Vettore disponibili

(a) Stato Iniziale

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
6	2	3

Vettore disponibili

(b) P2 Completa l'esecuzione

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
7	2	3

Vettore disponibili

(c) P1 Completa l'esecuzione

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
9	3	4

Vettore disponibili

(d) P3 Completa l'esecuzione

Figura 6.6 *Determinazione di uno stato sicuro*

richiesta. Comunque, tornando allo stato della Figura 6.7a, si supponga che P1 richieda un'unità di R1 e una di R3, e che le richieste siano accettate: lo stato risultante è quello della Figura 6.7b; è uno stato sicuro? La risposta è no, perché ogni processo avrà bisogno di almeno un'ulteriore unità di R1, e nessuna unità è disponibile. Quindi, sulla base dell'esclusione dello stallo, la richiesta di P1 dovrebbe essere rifiutata e il processo bloccato.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
9	3	6

Vettore delle risorse

R1	R2	R3
1	1	2

Vettore disponibili

(a) Stato iniziale

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matrice delle richieste

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matrice di assegnazione

R1	R2	R3
0	1	1

Vettore disponibili

(b) P1 richiede una unità ciascuno di R1 e R3

Figura 6.7 *Determinazione di uno stato non sicuro*

È importante sottolineare che la Figura 6.7b non rappresenta uno stato in stallo, ma uno in cui lo stallo è possibile. Ad esempio P1 potrebbe rilasciare un'unità di R1 e una di R3, per poi richiederle nuovamente quando ne avrà bisogno; in tal caso il sistema ritornerebbe in uno stato sicuro. Quindi la strategia di esclusione dello stallo non fa una previsione certa, ma semplicemente anticipa la possibilità di avere stallo, e garantisce che questo non avvenga.

La Figura 6.8 mostra una versione astratta della logica dell'esclusione dello stallo: l'algoritmo principale è nella parte (b). La struttura dati *stato* rappresenta lo stato del sistema, e *richieste_ora*[*] è il vettore delle richieste del processo *i*. Prima si effettua un controllo per assicurare che le richieste non siano superiori alla dichiarazione iniziale del processo; se la richiesta è valida, il passo successivo è verificare se è possibile soddisfarla (cioè se ci sono sufficienti risorse disponibili), e in caso negativo il processo è sospeso. Se è possibile, il passo finale consiste nel determinare se la concessione delle richieste porterebbe ad uno stato sicuro; per fare ciò calcola lo stato *nuovostato*, ottenuto assegnando le risorse al processo *i*, e si usa l'algoritmo della Figura 6.8c per stabilire se tale stato è sicuro.

Il vantaggio dell'esclusione dello stallo è che non è necessario interrompere i processi e riportarli ad uno stato precedente, cosa che avviene nel rilevamento dello stallo, ed è meno restrittiva della prevenzione dello stallo, in ogni modo impone alcune restrizioni:

- Ogni processo deve dichiarare all'inizio il numero massimo di risorse di cui avrà bisogno.
- I processi considerati devono essere indipendenti, cioè l'ordine di esecuzione non deve essere vincolato dalle esigenze di sincronizzazione.

```
type stato = record
              risorse, disponibili: array [0...m-1] of integer;
              richieste, assegnate: array [0.. .n-1, 0.. .m-1] of integer
            end
```

(a) Strutture dati globali

```
if assegnate[i,*] +  richieste_ora[*]  > richieste[i,*] then  < errore >
-richieste > richieste totali
else
    if richieste_ora[*] > disponibili[*] then  < sospendi il  processo >
    else                  -simula l'assegnazione
      < definisci nuovostato come:
      assegnate[i,*] := assegnate[i,*] + richieste_ora[*]
      disponibili [*] := disponibili [*] - richieste_ora[*]] >
   end;
   if sicuro(nuovostato) then
      < effettua l'assegnazione >
   else
      < ripristina lo stato originale >;
      < sospendi il processo >
   end
end.
```

(b) Algoritmo di assegnazione delle risorse

```
function sicuro (stato: S): boolean;
var  disp_corrente: array [0.. .m-1] of integer;
     resto: set of process;
begin
   disp_corrente := disponibili;
   resto := {tutti i processi};
    possibile := true;
   while possibile do
      trova un Pk in resto tale che
         richieste[k,*] - assegnate [k,*] ≤ disp_corrente;
      if trovato then          -simula l'esecuzione di P
         disp_corrente := disp_corrente + assegnate[k,*];
         resto := resto - {Pk}
      else
         possibile := false
   end;
   sicuro := (resto = null)
end.
```

(c) Test di sicurezza (algoritmo del banchiere)

Figura 6.8 *Logica di esclusione dello stallo*

- Ci deve essere un numero fissato di risorse da allocare.
- Nessun processo può terminare mentre possiede delle risorse.

6.4 Rilevamento dello stallo

Le strategie per la prevenzione dello stallo sono molto conservative: risolvono il problema limitando l'accesso alle risorse e imponendo delle restrizioni ai processi. All'estremo opposto, le strategie di rilevamento dello stallo non limitano l'accesso alle risorse e non impongono restrizioni alle azioni dei processi. Con il rilevamento dello stallo, quando è possibile le richieste vengono sempre soddisfatte, e il sistema operativo esegue periodicamente un algoritmo per rilevare la presenza della condizione di attesa circolare, descritta in precedenza nella condizione (4) e illustrata nella Figura 6.5.

L'algoritmo di rilevamento dello stallo

Si può effettuare un controllo sulla presenza di stallo ad ogni richiesta di risorsa, oppure meno frequentemente, a seconda della probabilità dello stallo. Controllare ogni richiesta ha due vantaggi: la rilevazione è precoce e l'algoritmo è relativamente semplice, perché è basato su modifiche incrementali dello stato del sistema. D'altro canto, questi controlli frequenti utilizzano parecchio tempo di processore.

Un comune algoritmo per il rilevamento dello stallo è quello descritto in [COFF71], che usa la matrice Assegnazione e il vettore Disponibili descritti nella sezione precedente. Inoltre si definisce una matrice Q, tale che q_{ij} rappresenta il numero di risorse di tipo j richieste dal processo i. L'algoritmo procede marcando i processi che non sono in stallo; all'inizio nessun processo è marcato. In seguito si effettuano i seguenti passi:

1. Si marca ogni processo che ha una riga di tutti zero nella matrice Assegnazione.
2. Si inizializza il vettore **W** con il contenuto del vettore Disponibili.
3. Si cerca un indice i tale che il processo i non è marcato e la i-esima riga di Q è minore o uguale a **W**, cioè $Q_{ik} \leq W_k$ per $1 \leq k \leq m$. Se una tale riga non esiste, l'algoritmo termina.
4. Se si trova tale riga, il processo i viene marcato e si aggiunge a **W** la riga corrispondente della matrice Assegnazione, cioè si assegna $W_k = W_k + A_{ik}$, e si ritorna al passo 3.

Si ha stallo se e solo se alla fine dell'algoritmo uno dei processi non è marcato: ogni processo non marcato è in stallo. La strategia di questo algoritmo è di trovare un processo le cui richieste possono essere soddisfatte con le risorse utilizzabili, supporre che queste richieste siano soddisfatte e che il processo completi l'esecuzione, liberando tutte le risorse. L'algoritmo poi cerca un altro processo da soddisfare. Si noti che l'algoritmo non garantisce la prevenzione dello stallo, che dipende dall'ordine in cui le richieste sono soddisfatte; tutto quello che fa è determinare se al momento c'è stallo

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Matrice delle richieste totali

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Matrice di assegnazione

R1	R2	R3	R4	R5
2	1	1	2	1

Vettore delle risorse

R1	R2	R3	R4	R5
0	0	0	0	1

Vettore disponibili

Figura 6.9 *Esempio di rilevamento dello stallo*

La Figura 6.9 illustra l'algoritmo di rilevamento dello stallo, che procede in questo modo:

- P4 viene marcato, perché non ha risorse assegnate.
- **W** viene messo a (0 0 0 0 1).
- La richiesta del processo P3 è minore o uguale a **W**, quindi P3 viene marcato e si assegna **W** = **W** + (0 0 0 1 0) = (0 0 0 1 1).
- L'algoritmo termina.

L'algoritmo termina con P1 e P2 non marcati, indicando che i due processi sono in stallo.

Ripristino

Quando lo stallo è stato rilevato, serve una strategia di ripristino; ci sono vari approcci possibili, in ordine di complessità crescente:

1. Tutti i processo in stallo vengono abortiti; questa è, che ci si creda o no, una delle soluzioni più comunemente adottate nei sistemi operativi, se non la più comune.
2. Ripristinare tutti i processi in stallo a uno stato precedente (*checkpoint*) e riattivarli. Questo richiede di incorporare nel sistema i meccanismi di ripristino e riattivazione; il rischio è che lo stallo originale capiti ancora, comunque il nondeterminismo dei processi concorrenti dovrebbe assicurare che questo non avvenga.
3. Abortire i processi in stallo uno dopo l'altro fino all'eliminazione dello stallo; l'ordine in cui si scelgono i processi da abortire dovrebbe basarsi su un criterio di costo minimo. Dopo ogni aborto è necessario attivare nuovamente l'algoritmo di rilevamento per vedere se c'è ancora stallo.
4. Revocare in sequenza le risorse fino all'eliminazione dello stallo. Come in 3, bisogna effettuare una selezione in base al costo, e bisogna riattivare l'algoritmo dopo ogni revoca. Un processo a cui si revoca una risorsa deve essere riportato ad uno stato precedente all'acquisizione di quella risorsa.

Per 3 e 4, i criteri di scelta potrebbero essere i seguenti: si sceglie il processo che

- ha consumato meno tempo di processore
- ha prodotto minor output
- ha il più alto tempo di esecuzione rimanente stimato
- ha allocato meno risorse
- ha priorità più bassa.

Alcune quantità sono di più facile misurazione di altre: stimare il tempo di esecuzione rimanente è particolarmente difficile. Inoltre, a parte la priorità, non c'è indicazione del "costo" per l'utente, a differenza del costo per l'intero sistema operativo.

6.5 Una strategia integrata per lo stallo

Come suggerito dalla Tabella 6.1, ogni strategia per la gestione dello stallo ha vantaggi e svantaggi; anziché cercare di progettare un sistema operativo che utilizza una sola di queste strategie, potrebbe essere più efficiente usare strategie diverse in situazioni diverse. [SILB94] suggerisce un approccio:

- Raggruppare le risorse in alcune classi di risorse.
- Usare la strategia di ordine lineare, definita precedentemente, per la prevenzione dell'attesa circolare in modo da prevenire lo stallo fra classi di risorse.
- All'interno di ogni classe di risorse, usare l'algoritmo più appropriato per quella classe.

Per vedere un esempio di questa tecnica, si considerino le seguenti classi di risorse:

- **Spazio di swap:** blocchi di memoria secondaria da usare per il trasferimento dei processi.
- **Risorse dei processi:** dispositivi che possono essere assegnati ai processi, come lettori di nastri e file.
- **Memoria principale:** assegnabile ai processi in pagine o segmenti.
- **Risorse interne:** ad esempio canali di I/O.

L'ordine della lista precedente rappresenta quello in cui le risorse sono assegnate; è un ordine ragionevole, considerando la sequenza di passi che un processo può seguire durante l'esecuzione. All'interno di ogni classe si potrebbero usare le seguenti strategie:

- **Spazio di swap:** prevenzione dello stallo imponendo che tutte le risorse siano richieste contemporaneamente, come nella strategia di prevenzione possesso-e-attesa. Questa strategia è ragionevole se, come spesso accade, si conoscono a priori le richieste massime; l'esclusione dello stallo è un'altra possibilità.

- **Risorse dei processi:** in questa categoria l'esclusione è spesso la strategia migliore, perché è ragionevole aspettarsi che i processi dichiarino all'inizio le risorse di questa classe di cui avranno bisogno. Un'altra scelta possibile è la prevenzione tramite ordinamento delle risorse all'interno della classe.
- **Memoria principale:** la prevenzione tramite prerilascio sembra la strategia più appropriata: quando bisogna liberare memoria in possesso di un processo, è sufficiente scaricare il processo su memoria secondaria e riutilizzare lo spazio per risolvere lo stallo.
- **Risorse interne:** si può usare la prevenzione tramite ordinamento delle risorse.

6.6 Il problema dei filosofi a tavola

C'erano una volta cinque filosofi che vivevano insieme; la vita di ogni filosofo consisteva principalmente in pensare e mangiare, e dopo anni di meditazione, tutti i filosofi erano giunti alla conclusione che l'unico cibo che stimolava i loro pensieri erano gli spaghetti.

L'organizzazione dei pasti era semplice (Figura 6.10): una tavola rotonda con un grande piatto di spaghetti al centro, cinque piatti, uno per ogni filosofo, e cinque forchette. Quando un filosofo voleva mangiare, doveva sedersi al proprio posto e, usando le due forchette ai lati del suo piatto, servirsi la pasta e mangiarla. Il problema è questo: trovare un rituale (algoritmo) che permetta ai filosofi di mangiare. L'algoritmo deve soddisfare la mutua esclusione (due filosofi

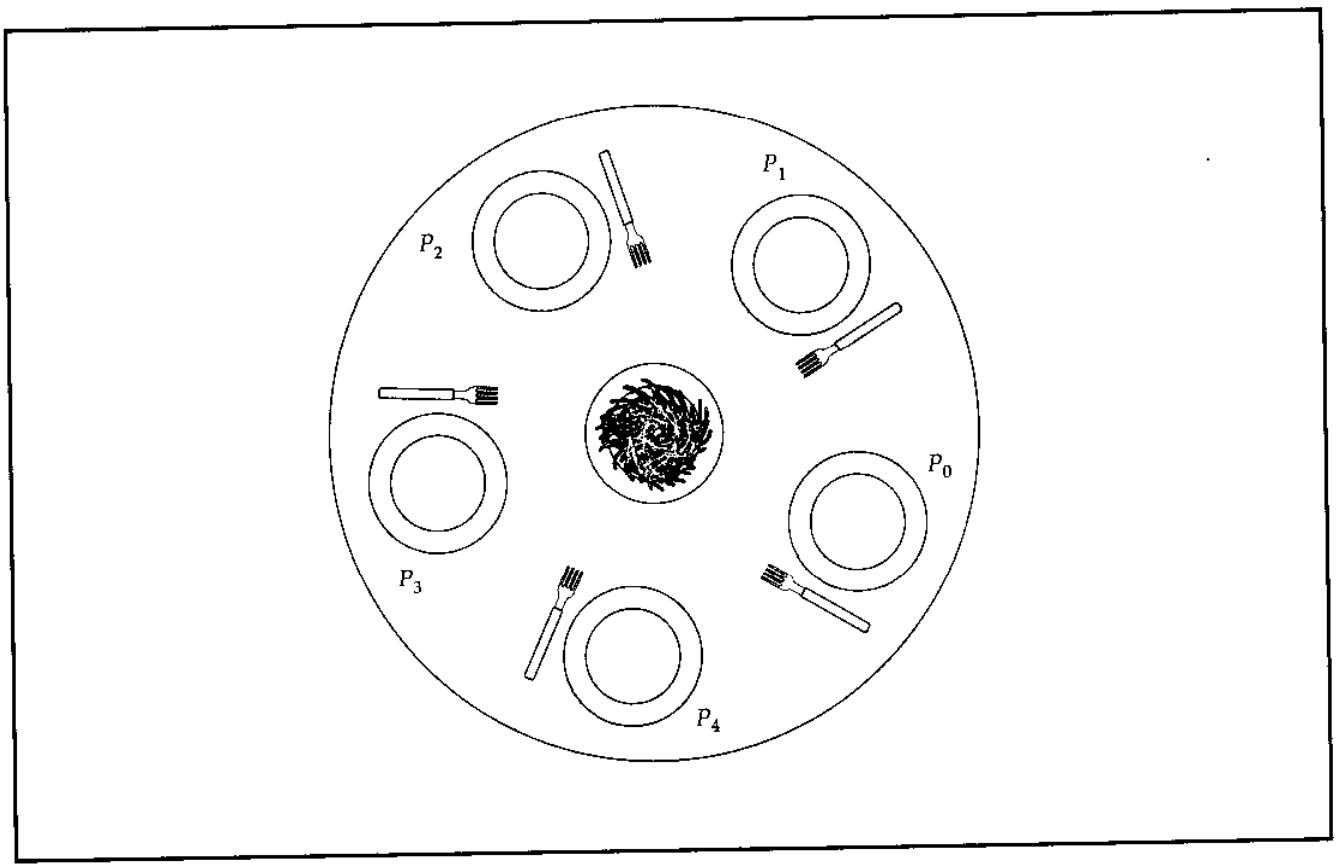


Figura 6.10 *La tavola dei filosofi*

```
program filosofi_a_tavola;
var     forchetta: array [0...4] of semaforo (:= 1);
        i: integer;
procedure filosofo (i: integer)
begin
    repeat
        pensa;
        wait ( forchetta[i] );
        wait (forchetta[(i+1) mod 5]);
        mangia;
        signal (forchetta[ (i+1) mod 5]);
        signal ( forchetta[i])
    forever
end;
begin
    parbegin
        filosofo(0);
        filosofo(1);
        filosofo(2);
        filosofo(3);
        filosofo(4)
    parend
end.
```

Figura 6.11 *Prima soluzione al problema dei filosofi a tavola*

non possono usare la stessa forchetta contemporaneamente) ed evitare lo stallo e la starvation (in questo caso, il significato letterale si sposa con quello algoritmico!).

Questo problema, dovuto a Dijkstra, a prima vista non sembra rilevante, comunque illustra i problemi di base dello stallo e della starvation; inoltre, i tentativi di sviluppare una soluzione rivelano molte delle difficoltà della programmazione concorrente (ad esempio si veda [GING90]). Quindi, questo problema è un test standard per valutare gli approcci alla sincronizzazione.

La Figura 6.11 suggerisce una soluzione con semafori. Ogni filosofo prende prima la forchetta di sinistra e poi quella di destra; dopo che un filosofo ha finito di mangiare, ripone le due forchette sulla tavola. Questa soluzione, purtroppo, può portare a stallo: se tutti i filosofi vogliono mangiare nello stesso momento, si siedono, ognuno prende la forchetta alla propria sinistra e poi tenta di prendere quella di destra, che non c'è. In questa situazione poco dignitosa tutti i filosofi muoiono di fame.

Per superare il rischio di stallo, si potrebbero comprare altre cinque forchette (una soluzione più igienica!), o si potrebbe insegnare ai filosofi a mangiare gli spaghetti con una sola forchetta. Un altro approccio potrebbe essere quello di aggiungere un cameriere che permette l'accesso alla sala da pranzo ad un massimo di quattro filosofi: in tal caso, almeno un filosofo ha accesso a due forchette. La Figura 6.12 mostra questa soluzione, sempre con semafori, che non ha il problema dello stallo né quello della starvation.

```
program filosofi_a_tavola;
var     forchetta: array [0.. .4] of semaforo (:= 1);
        stanza: semaforo (:= 4);
        i: integer;
procedure filosofo (i: integer);
begin
    repeat
        pensa;
        wait (stanza);
        wait ( forchetta[i] );
        wait (forchetta[(i+1) mod 5])
         mangia;
        signal (forchetta [(i+1) mod 5]);
        signal ( forchetta[i]);
        signal (stanza)
    forever
end;
begin
    parbegin
        filosofo(0);
        filosofo(1);
        filosofo(2);
        filosofo(3);
        filosofo(4)
    parend
end.
```

Figura 6.12 *Seconda soluzione al problema dei filosofi a tavola*

6.7 I meccanismi di UNIX per la concorrenza

UNIX fornisce vari meccanismi per la comunicazione tra processi e per la sincronizzazione. Si esaminano di seguito i più importanti:

- Pipe
- Messaggi
- Memoria condivisa
- Semafori
- Segnali.

Le pipe, i messaggi e la memoria condivisa forniscono un sistema di comunicazione di dati fra processi, mentre i semafori e i segnali sono usati per attivare azioni di altri processi.

Pipe

Le pipe sono uno dei contributi più significativi di UNIX allo sviluppo dei sistemi operativi. Ispirata dal concetto di coroutine [RITC84], una pipe è un buffer circolare che permette a due processi di comunicare sul modello del produttore/consumatore, quindi è una coda first-in-first-out, scritta da un processo e letta da un altro.

Una pipe è creata con una dimensione fissata di byte; quando un processo cerca di scrivere nella pipe, la scrittura è effettuata immediatamente se c'è abbastanza spazio, altrimenti il processo è bloccato. Analogamente un processo che legge dalla pipe è bloccato se cerca di leggere più dati di quanti la pipe ne contenga, altrimenti la lettura è eseguita immediatamente. Il sistema operativo garantisce la mutua esclusione: un solo processo alla volta può accedere alla pipe.

Ci sono due tipi di pipe: con e senza nome. Solo due processi imparentati possono condividere una pipe senza nome, mentre i processi senza relazioni di parentela possono condividere solo pipe con nome.

Messaggi

Un messaggio è un blocco di testo con associato un tipo; UNIX fornisce le chiamate di sistema *msgsnd* e *msgrcv* per il passaggio di messaggi fra processi. Ogni processo ha associata una coda di messaggi, che ha la funzione di casella postale.

Il mittente del messaggio deve specificare il tipo di ogni messaggio che vuole inviare, e questo può essere usato dal ricevente come criterio di selezione. Il ricevente può ricevere i messaggi in ordine first-in-first-out, oppure in base al tipo. Un processo è sospeso quando cerca di mandare un messaggio ad una coda piena, o quando cerca di leggere da una coda vuota. Se un processo cerca di leggere un messaggio di un certo tipo e fallisce, non viene sospeso.

Memoria condivisa

Il metodo più veloce per la comunicazione interprocesso in UNIX è l'uso della memoria condivisa: un blocco di memoria virtuale condiviso da vari processi, che effettuano letture e scritture utilizzando le stesse istruzioni macchina usate per le altre porzioni di memoria virtuale. Ogni processo può avere il permesso di sola lettura, oppure di lettura/scrittura. La mutua esclusione non è garantita dal gestore di memoria, ma è lasciata a carico dei processi che la usano.

Semafori

Le chiamate di sistema per i semafori in UNIX System V sono una generalizzazione delle primitive *wait* e *signal* descritte nel Capitolo 5, nel senso che molte operazioni possono essere effettuate contemporaneamente, e l'incremento e decremento del contatore può essere fatto con un valore maggiore di 1. Il kernel effettua tutte le operazioni in modo atomico: nessun altro processo può accedere al semaforo prima della conclusione delle operazioni.

Un semaforo consiste dei seguenti elementi:

- Valore corrente del semaforo.
- Identificatore dell'ultimo processo che ha avuto accesso al semaforo.
- Numero di processi in attesa che il valore attuale del semaforo venga incrementato.
- Numero di processi in attesa che il valore del semaforo diventi zero.

Associata ad ogni semaforo c'è una coda di processi sospesi su quel semaforo.

I semafori, in effetti, vengono creati a gruppi, dove ogni gruppo comprende uno o più semafori. C'è una chiamata di sistema *semctl*, che imposta tutti i valori dei semafori del gruppo contemporaneamente, e un'altra chiamata, *semop*, che prende come argomento una lista di operazioni sui semafori, ciascuna operante su uno dei semafori del gruppo: il kernel effettua le operazioni una alla volta. Per ogni operazione, la funzione è specificata dal valore *sem_op*; ci sono le seguenti possibilità:

- Se *sem_op* è positiva, il kernel incrementa il valore del semaforo e attiva tutti i processi in attesa di un incremento del semaforo.
- Se *sem_op* è 0, il kernel controlla il valore del semaforo; se è 0, continua con le altre operazioni della lista, altrimenti incrementa il numero dei processi in attesa che il semaforo abbia valore 0, e sospende il processo sull'evento che il valore del semaforo sia 0.
- Se *sem_op* è negativa e il suo valore assoluto è minore o uguale a quello del semaforo, il kernel aggiunge *sem_op* (un valore negativo) al valore del semaforo; se il risultato è 0, il kernel attiva tutti i processi che stavano aspettando che il valore del semaforo diventasse 0.
- Se *sem_op* è negativa e il suo valore assoluto è maggiore di quello del semaforo, il kernel sospende il processo sull'evento che il valore del semaforo venga incrementato.

Questa generalizzazione dei semafori fornisce una flessibilità considerevole per la sincronizzazione e la coordinazione dei processi.

Segnali

Un segnale è un meccanismo software che informa un processo dell'occorrenza di eventi asincroni; è simile ad un interrupt hardware ma non usa il concetto di priorità. Quindi tutti i segnali sono trattati nello stesso modo: anche se vengono mandati contemporaneamente, arrivano al processo uno alla volta, senza un ordine particolare.

I processi possono scambiarsi segnali, e il kernel può inviare segnali internamente; un segnale viene consegnato aggiornando un campo nella tabella del processo ricevente; poiché ogni segnale è rappresentato da un bit, i segnali non possono essere accodati. Ogni segnale è elaborato non appena il processo viene risvegliato, o quando si prepara a ritornare da una chiamata di sistema. Un processo può rispondere ad un segnale effettuando qualche azione per difetto (es. terminazione), eseguendo una funzione di gestione del segnale, o ignorandolo.

La Tabella 6.2 elenca i segnali definiti in UNIX SVR4.

Tabella 6.2 *Segnali di UNIX*

Valore	Nome	Descrizione
01	SIGHUP	hang up; mandato al processo quando il sistema operativo ritiene che l'utente del processo non stia facendo lavoro utile
02	SIGINT	interruzione
03	SIGQUIT	quit; mandato dall'utente per fermare un processo, e produrre una immagine del contesto del processo
04	SIGILL	istruzione illegale
05	SIGTRAP	provoca l'esecuzione di una trap per seguire la traccia del processo
06	SIGIOT	istruzione IOT
07	SIGEMT	istruzione EMT
08	SIGFPT	eccezione floating-point
09	SIGKILL	kill; termina il processo
10	SIGBUS	errore del bus
11	SIGSEGV	violazione del segmento; il processo cerca di accedere ad una locazione fuori dal proprio spazio di indirizzamento virtuale
12	SIGSYS	chiamata di sistema con argomento errato
13	SIGPIPE	scrittura su una pipe che non ha lettori associati
14	SIGALARM	sveglia; mandato quando un processo vuole ricevere un segnale dopo un certo periodo di tempo
15	SIGTERM	terminazione software
16	SIGUSR1	segnale definito dall'utente numero 1
17	SIGUSR2	segnale definito dall'utente numero 2
18	SIGCLD	morte del processo figlio
19	SIGPWR	interruzione dell'alimentazione

6.8 Primitive per la sincronizzazione dei thread in Solaris

Oltre ai meccanismi di concorrenza di UNIX SVR4, Solaris supporta altre quattro primitive di sincronizzazione:

- Lock di mutua esclusione (*mutex*)
- Semafori
- Lock per lettori multipli e scrittore singolo (lettori/scrittori)
- Variabili di condizione.

Solaris implementa queste primitive all'interno del kernel nel caso di thread del kernel, ma anche nella libreria dei thread, nel caso di thread a livello di utente. L'esecuzione di una primitiva crea una struttura dati che contiene parametri specificati dal thread che la chiama (Figura 6.13). Quando un oggetto di sincronizzazione viene creato, si possono effettuare essenzialmente due operazioni: *enter* (acquisizione, lock) e *release* (rimozione del lock). Nel kernel e nella libreria dei thread non ci sono meccanismi per garantire la mutua esclusione o per prevenire lo stallo. Se un thread cerca di accedere ad una parte di codice o di dati che dovrebbero essere protetti, ma non usa le primitive di sincronizzazione appropriate, l'accesso viene effettuato. Se un thread blocca un oggetto e poi dimentica di sbloccarlo, il kernel non effettua alcun'azione.

Tutte le primitive di sincronizzazione richiedono l'esistenza di un'istruzione hardware per controllare un oggetto ed assegnare ad esso un valore in un'operazione atomica, come visto nella Sezione 5.3.

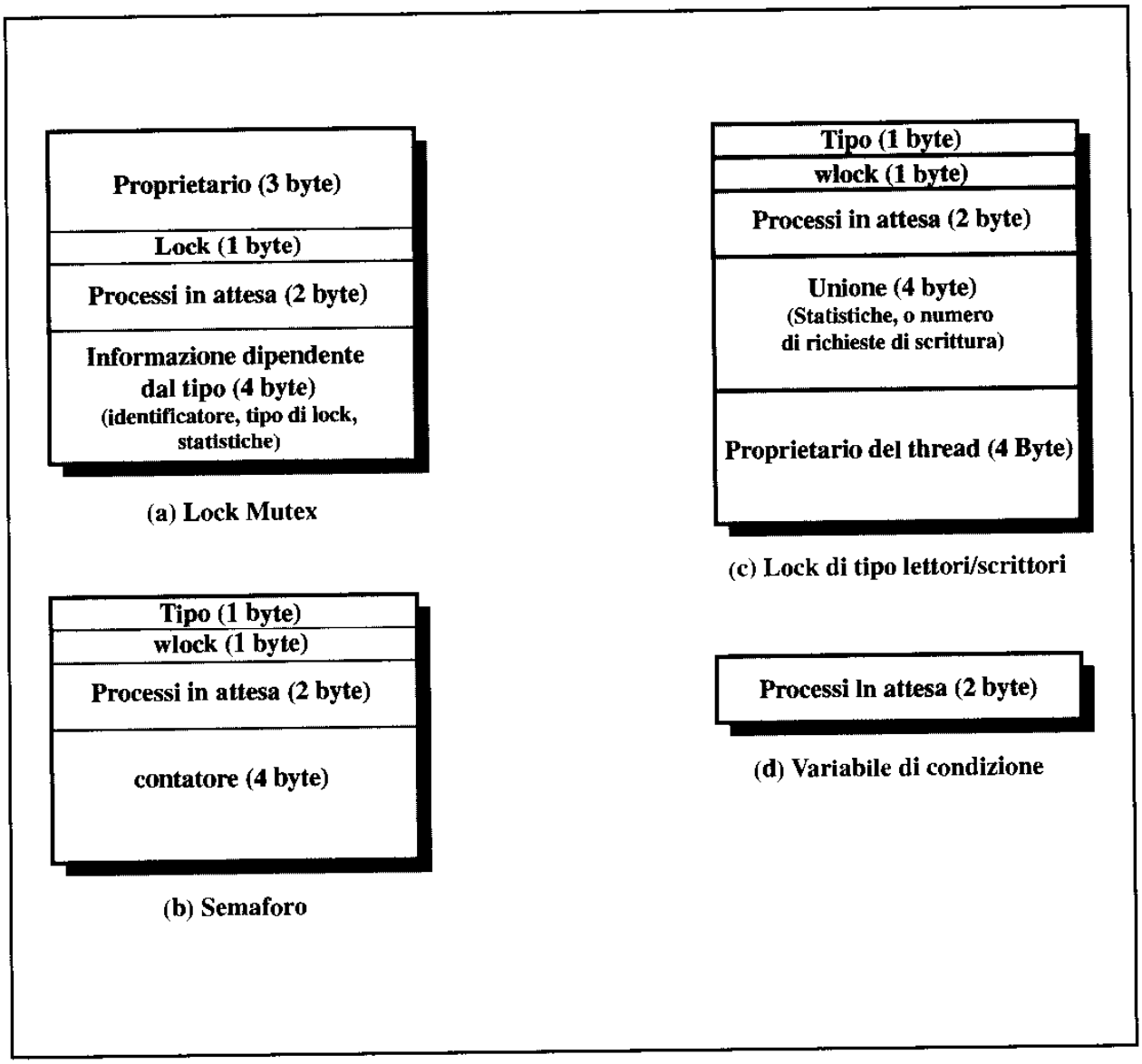


Figura 6.13 *Strutture dati per la sincronizzazione in Solaris*

Lock per la mutua esclusione

Un lock di tipo mutex (mutua esclusione) impedisce che più di un processo prosegua una volta acquisito il lock; il thread che blocca il mutex deve essere anche quello che lo sblocca. Un thread cerca di acquisire un lock tramite la primitiva *mutex_enter*; se questa non può creare il lock (perché è già stato creato da un altro processo), l'azione effettuata dipende dall'informazione di tipo contenuta nell'oggetto mutex. La politica di bloccaggio standard è che un thread bloccato controlla periodicamente lo stato del lock tramite un ciclo di attesa (*spin lock*). Si può avere anche un meccanismo di bloccaggio basato sugli interrupt: in tal caso, il mutex contiene un identificatore della coda di thread in attesa su quel lock.

Le primitive associate ad un lock di tipo mutex sono le seguenti:

`mutex_enter()`	Acquisisce il lock, bloccando il richiedente se questo esiste già.
`mutex_exit()`	Rilascia il lock, magari sbloccando un processo in attesa.
`mutex_tryenter()`	Acquisisce il lock se non esiste già.

La primitiva *mutex_tryenter()* fornisce un modo per realizzare la mutua esclusione senza effettuare blocchi; il programmatore può così usare un approccio di attesa attiva per i thread a livello di utente, evitando di bloccare l'intero processo perché un thread è bloccato.

Semafori

Solaris fornisce i semafori standard, con le primitive seguenti:

`sema_p()`	Decrementa il semaforo, magari bloccando il thread.
`sema_v()`	Incrementa il semaforo, magari sbloccando un processo in attesa.
`sema_tryp()`	Decrementa il semaforo se non è necessario bloccare il thread.

Come prima, la primitiva *sema_tryp()* permette l'attesa attiva.

Lock di tipo lettori/scrittore

Un lock di tipo lettori/scrittore permette a molti thread di accedere contemporaneamente in sola lettura ad un oggetto protetto dal lock, e ad un singolo thread di scrivere, escludendo tutti i lettori. Quando si acquisisce il lock in scrittura, l'oggetto entra in stato *lock di scrittura*: tutti i thread che cercano di accedere in lettura o scrittura devono aspettare. Se uno o più lettori hanno acquisito il lock, l'oggetto è in stato *lock di lettura*. Le primitive sono le seguenti:

`rw_enter()`	Cerca di acquisire un lock di lettura o scrittura.
`rw_exit()`	Rilascia un lock di lettura o scrittura.

`rw_tryenter()`	Acquisisce il lock se il blocco non è necessario.
`rw_downgrade()`	Un thread che ha acquisito un lock di scrittura lo converte in lock di lettura. Ogni scrittore in attesa vi rimane, finché questo thread non rilascia il lock. Se non ci sono scrittori in attesa, la primitiva sveglia tutti i lettori in attesa.
`rw_tryupgrade()`	Cerca di convertire un lock di lettura in un lock di scrittura.

Variabili di condizione

Le variabili di condizione sono usate per aspettare il verificarsi di una certa condizione; vanno usate insieme con un lock di tipo mutex. Ciò implementa un monitor del tipo illustrato nella Figura 5.22. Le primitive sono le seguenti:

`cv_wait()`	Aspetta di ricevere il segnale indicante il verificarsi della condizione.
`cv_signal()`	Riattiva uno dei thread bloccati in una *cv_wait()*.
`cv_broadcast()`	Riattiva tutti i thread bloccati in una *cv_wait()*.

cv_wait() rilascia il relativo mutex prima di bloccarsi e lo acquisisce nuovamente prima di ritornare; poiché la successiva acquisizione del mutex può essere bloccata da altri thread in attesa su di esso, la condizione che ha causato l'attesa deve essere controllata nuovamente. Quindi l'uso tipico è il seguente:

```
mutex_enter(&m)
...
while (qualche_condizione) {
   cv_wait(&cv, &m);
}
...
mutex_exit(&m);
```

Ciò permette di usare un'espressione complessa come condizione, perché è protetta dal mutex.

6.9 I meccanismi di Windows NT per la concorrenza

In Windows NT la sincronizzazione fra i thread fa parte dell'architettura degli oggetti; viene implementata tramite la seguente famiglia di oggetti di sincronizzazione:

- Processi
- Thread

Tabella 6.3 *Oggetti di sincronizzazione di Windows NT*

Tipo di oggetto	Definizione	Messo in stato signaled quando	Effetto sui thread in attesa
Processo	Esecuzione di un programma, con spazio di indirizzamento e risorse necessarie all'esecuzione	L'ultimo thread termina	Tutti rilasciati
Thread	Entità eseguibile all'interno di un processo	Il thread termina	Tutti rilasciati
File	Istanza di un file aperto o dispositivo di I/O	L'operazione di I/O è terminata	Tutti rilasciati
Input dalla console	Buffer per la finestra di testo (es. usato per gestire l'I/O su schermo di un'applicazione MS-DOS)	L'input è disponibile per elaborazione	Un thread viene rilasciato
Notifica del cambiamento di un file	Notifica di un cambiamento nel file system	Avviene un cambiamento nel file system che rispetta i criteri di filtro di questo oggetto	Un thread viene rilasciato
Mutex	Meccanismo per la mutua esclusione in Win32 e OS/2	Il thread che lo possiede, o un altro thread, lo rilascia	Un thread viene rilasciato
Semaforo	Contatore che regola il numero di thread che usano una risorsa	Il contatore diventa zero	Tutti rilasciati
Evento	Annuncio di un evento di sistema	Il thread setta l'evento	Tutti rilasciati
Temporizzatore delle attese	Contatore che registra il passaggio del tempo	Arriva l'ora indicata, o l'intervallo di tempo è passato	Tutti rilasciati

Nota: le righe grigie corrispondono agli oggetti che servono solo per la sincronizzazione.

- File
- Input dalla console
- Notifica della modifica dei file
- Mutex
- Semafori
- Eventi
- Temporizzatore delle attese.

I primi quattro tipi di oggetti della lista precedente hanno anche altri usi, ma possono essere impiegati per la sincronizzazione; i rimanenti sono progettati espressamente per la sincronizzazione.

Ogni istanza di un oggetto di sincronizzazione può essere in uno stato segnalato o non segnalato. Un thread può essere sospeso su un oggetto in stato non segnalato; il thread viene rilasciato quando l'oggetto passa nello stato segnalato. Il meccanismo è banale: un thread manda una richiesta di attesa ad NT Executive, utilizzando lo handle dell'oggetto di sincronizzazione. Quando un oggetto entra in stato segnalato, NT Executive rilascia tutti i thread in attesa su quell'oggetto.

La Tabella 6.3 riassume, per ogni tipo, gli eventi che fanno passare un oggetto nello stato segnalato e l'effetto che ciò ha sui thread in attesa.

L'oggetto mutex è usato per garantire la mutua esclusione nell'accesso ad una risorsa, permettendo ad un solo thread alla volta di avere l'accesso; quindi funziona come un semaforo binario. Quando l'oggetto mutex entra nello stato segnalato, viene rilasciato uno solo fra i thread in attesa sul mutex. I mutex possono essere usati per sincronizzare thread appartenenti a processi diversi.

Come i mutex, anche i semafori, che in NT sono implementati nel modo classico, a contatore, possono essere condivisi da thread di processi diversi.

Il temporizzatore delle attese è un nuovo oggetto del kernel di NT 4.0, che manda segnali ad un'ora particolare e/o ad intervalli regolari.

6.10 Sommario

Lo stallo è il blocco di un insieme di processi che competono per l'accesso alle risorse di sistema, o comunicano fra loro. Il blocco è permanente a meno che il sistema operativo non effettui delle azioni straordinarie, come l'eliminazione di uno o più processi, o il ripristino di qualche processo ad uno stato precedente. Le risorse che scatenano lo stallo si dividono in riutilizzabili e non riutilizzabili: le seconde sono quelle che vengono distrutte dopo l'uso, come i messaggi e i dati nei buffer di I/O; le prime sono quelle che non vengono distrutte, come i canali di I/O e le regioni di memoria.

Ci sono tre approcci generali alla gestione dello stallo: prevenzione, rilevamento e esclusione. La prevenzione dello stallo garantisce che lo stallo non capiterà, assicurandosi che una delle

condizioni necessarie per lo stallo non si verifichi. Il rilevamento dello stallo è necessario, se si vuole che il sistema operativo soddisfi sempre le richieste di risorse; il sistema controllerà periodicamente la presenza di stallo, ed effettuerà le operazioni necessarie per eliminarlo. L'esclusione dello stallo consiste nell'analisi di ogni nuova richiesta di risorse, per determinare se questa può portare ad uno stallo, e soddisfacendola solamente se lo stallo non è possibile.

6.11 Letture raccomandate

Il riferimento classico per lo stallo, [HOLT72], è ancora valido, così come [COFF71]. Un altro buon lavoro di rassegna è [ISLO80]. [CORB96] è uno studio accurato del rilevamento dello stallo.

I meccanismi per la concorrenza di UNIX SVR4, Solaris 2.x e Windows NT 4.0 sono trattati rispettivamente in [GRAY97], [GRAH95] e [RICH97].

CORB96 Corbett J. "Evaluating Deadlock Detection Methods for Concurrent Software". *IEEE Transactions on Software Engineering*, Marzo 1996.

COFF71 Coffman E., Elphick M. e Shoshani A. "System Deadlocks". *Computing Surveys*. Giugno 1971.

GRAH95 Graham J. *Solaris 2.x: Internals and Architecture*. New York: McGraw-Hill, 1995.

GRAY97 Gray J. *Interprocess Communications in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

HOLT72 Holt R. "Some Deadlock Properties of Computer Systems". *Computing Surveys*, Settembre 1972.

ISLO80 Isloor S. e Marsland T. "The Deadlock Problem: An Overview". *Computer*, Settembre 1980.

RICH97 Richter J. *Advanced Windows*. Redmond, WA: Microsoft Press, 1997.

6.12 Problemi

6.1 Si spieghino a parole i sei percorsi della figura 6.3, analogamente a quanto fatto nella Sezione 6.1 per la Figura 6.2.

6.2 Si consideri la seguente descrizione di un sistema, in cui non ci sono code di richieste pendenti.

disponibili

r1	r2	r3	r4
2	1	0	0

	assegnazione corrente				richiesta massima				richieste residue			
processo	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2				
p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

a. Calcolare il numero di risorse che ogni processo può ancora richiedere, completando le colonne "richieste residue".

b. Questo sistema è in uno stato sicuro? Perché?

c. È in stallo? Perché?

d. Ci sono dei processi in stallo, o che potrebbero entrarvi?

e. Se arriva una richiesta di (0, 1, 0, 0) da P3, è sicuro soddisfarla immediatamente? In quale stato (stallo, sicuro, non sicuro) entra il sistema se si soddisfa subito tutta la richiesta? Ci sono dei processi che sono o potrebbero entrare in stallo se la richiesta fosse esaudita immediatamente?

6.3 Si applichi l'algoritmo di rilevamento dello stallo della Sezione 6.4 ai dati seguenti e si mostrino i risultati.

$$\text{Disponibili} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

$$\text{Richieste} = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

$$\text{Assegnazione} = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

6.4 Uno spooler (Figura 6.4) consiste di un processo I per l'input, un processo utente P e un processo O per l'output, connessi da due buffer. I processi si scambiano blocchi di dati di dimensione fissata. I blocchi vengono scritti su disco, con un confine variabile fra buffer di input e di output, a seconda della velocità dei processi. Le primitive di comunicazione assicurano che il seguente vincolo sulle risorse sia soddisfatto:

$$i + o \leq max$$

dove

max = numero massimo di blocchi sul disco

i = numero di blocchi di input sul disco

o = numero di blocchi di output sul disco

Si tenga presente che:

a. Se l'ambiente fornisce dei dati, il processo I prima o poi li scriverà sul disco (se c'è spazio sufficiente).
b. Se il disco contiene dei dati di input, il processo P prima o poi li consumerà e scriverà sul disco una quantità finita di dati per ogni blocco di input (se c'è spazio sufficiente).
c. Finché il disco contiene dei dati di output, il processo O prima o poi li consumerà.

Si dimostri che in questo sistema si può avere stallo.

6.5 Si suggerisca un ulteriore vincolo sulle risorse del Problema 6.4 in modo che non si possa avere stallo, ma che si possa ancora avere un confine variabile fra i buffer a seconda delle necessità dei processi.

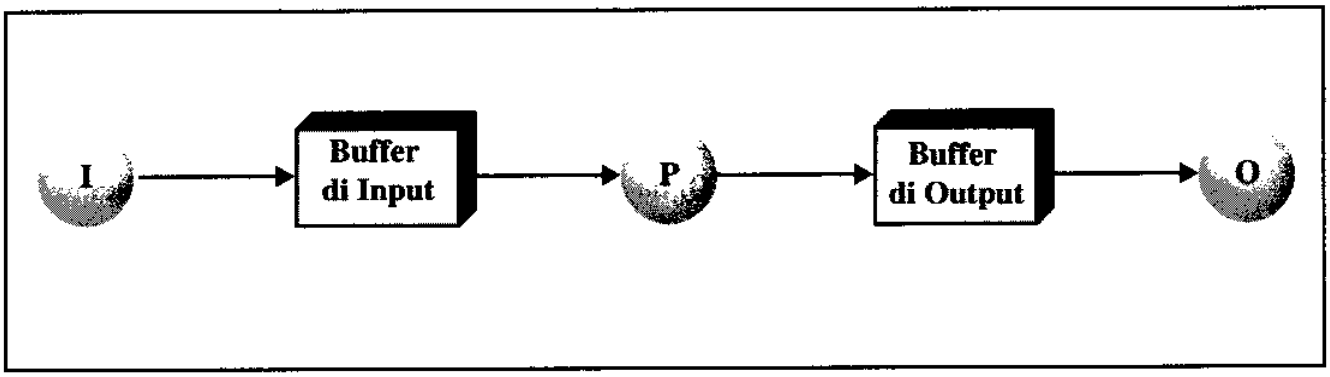


Figura 6.14 *Uno spooler*

6.6 Nel sistema di multiprogrammazione THE [DIJK68], un tamburo (il precursore del disco come dispositivo di memoria secondaria) è diviso in buffer di input, aree di calcolo, buffer di output, con confini variabili, a seconda della velocità dei processi coinvolti. Lo stato corrente del tamburo è caratterizzato dai parametri seguenti:

max = numero massimo di pagine sul tamburo
i = numero di pagine di input sul tamburo
p = numero di pagine di calcolo sul tamburo

o = numero di pagine di output sul tamburo
riso = minimo numero di pagine riservate per l'output
risp = minimo numero di pagine riservate per il calcolo

Si scrivano i vincoli sulle risorse necessari a garantire che la capacità del tamburo non sia superata, e che un numero minimo di pagine sia sempre riservato per l'output.

6.7 Nel sistema multiprogrammato THE, una pagina può effettuare le seguenti transizioni di stato:

1. vuoto → buffer di input (produzione di input)
2. buffer di input → area di calcolo (utilizzo dell'input)
3. area di calcolo → buffer di output (produzione dell'output)
4. buffer di output → vuoto (utilizzo dell'output)
5. vuoto → area di calcolo (chiamata di procedura)
6. area di calcolo → vuoto (ritorno da chiamata)

a. Si descriva l'effetto di queste transizioni sulle quantità *i*, *o* e *p*.

b. Ci sono delle transizioni che possono portare a stallo, sotto le ipotesi del Problema 6.4 sui processi di input, di output e utente?

6.8 Si consideri un sistema con 150 unità di memoria, assegnate a tre processi in questo modo:

Processo	Massimo	Attuale
1	70	45
2	60	40
3	60	15

Si applichi l'algoritmo del banchiere per determinare se sarebbe sicuro soddisfare ciascuna delle richieste seguenti. Se sì, indicare una sequenza di processi che possono terminare, altrimenti mostrare la riduzione della tabella di assegnazione risultante.

a. Arriva un quarto processo, con una richiesta iniziale di 25 unità di memoria e massima di 60.

b. Arriva un quarto processo, con una richiesta iniziale di 35 unità di memoria e massima di 60.

6.9 Si valuti l'utilità dell'algoritmo del banchiere nella vita reale.

6.10 Un algoritmo per una pipeline è implementato in modo tale che una sequenza di elementi di tipo T prodotti da un processo P_0 passi attraverso una serie di processi $P_1, P_2, ..., P_{n-1}$, che operano sugli elementi nell'ordine.

a. Si definisca un buffer generalizzato per i messaggi che contenga tutti i dati parzialmente consumati, e si scriva un algoritmo per il processo $P_i (0 \leq i \leq n-1)$ della forma

```
repeat
   ricevi dal precedente;
   consuma l'elemento;
   manda al successivo
forever
```

Si supponga che P_0 riceva elementi vuoti mandati da P_{n-1}. L'algoritmo dovrebbe permettere ai processi di operare direttamente sui messaggi memorizzati nel buffer, in modo da evitare di doverli copiare.

b. Si dimostri che i processi non possono essere in stallo rispetto al buffer comune.

6.11 **a.** Tre processi condividono quattro unità di una risorsa; si può riservare o rilasciare una sola unità alla volta. Ogni processo ha bisogno al massimo di due unità. Si dimostri che non si può avere stallo.

b. *N* processi condividono *M* unità di una risorsa; si può riservare o rilasciare una sola unità alla volta. Ogni processo ha bisogno al massimo di *M* unità o meno, e la somma delle richieste massime è minore di $N + M$. Si dimostri che non si può avere stallo.

6.12 Si considerino i modi seguenti di gestire lo stallo: (1) l'algoritmo del banchiere, (2) rilevare lo stallo e far terminare il thread rilasciandone le risorse, (3) riservare tutte le risorse in anticipo, (4) riavviare il thread e rilasciarne tutte le risorse se questo deve aspettare, (5) ordinare le risorse e (6) rilevare lo stallo e ripristinare il thread ad uno stato precedente.

a. Un criterio da usare per valutare i diversi approcci allo stallo è il seguente: quale permette un maggior livello di concorrenza? In altre parole, quale permette al maggior numero di thread di procedere senza dover aspettare quando non c'è stallo? Si assegni un numero da 1 a 6 a ognuno degli approcci elencati, dove 1 permette il massimo grado di concorrenza, commentando i risultati.

b. Un altro criterio è l'efficienza; in altre parole, quale approccio richiede minor tempo di processore? Si assegni un numero da 1 a 6 a ognuno degli approcci elencati, dove 1 permette l'efficienza massima, commentando i risultati. Cosa cambierebbe nell'ordine sapendo che gli stalli sono frequenti?

6.13 Si commenti la seguente soluzione al problema dei filosofi a tavola. Un filosofo affamato prende prima la forchetta di sinistra; se anche quella di destra è disponibile la prende e inizia a mangiare, altrimenti posa la forchetta di sinistra e ripete il ciclo.

6.14 Questo problema dimostra che il problema dei filosofi a tavola è subdolo, e che è difficile scrivere programmi corretti usando i semafori. La Figura 6.15 mostra una soluzione del problema dei filosofi a tavola presente in due testi recenti di sistemi operativi ([TANE97] e [TANE92]).

a. Si descriva a parole il funzionamento di questa soluzione.

b. L'autore dichiara: "La soluzione .... è corretta e permette il parallelismo massimo per un numero arbitrario di filosofi". Anche se la soluzione previene lo stallo, non è cor-

```
#define N                    5              /* numero di filosofi */
#define SINISTRA             (i-1) mod N    /* numero del vicino sinistro di i */
#define DESTRA               (i+1) mod N    /* numero del vicino destro di i */
#define PENSA                0              /* il filosofo sta pensando*/
#define AFFAMATO             1              /* il filosofo cerca di prendere le forchette */
#define MANGIA               2              /* il filosofo sta mangiando */
typedef int semaforo;                       /* i semafori sono un tipo speciale di interi */
int stato[N];                               /* array per tenere traccia dello stato di ciascuno */
semaforo mutex = 1;                         /* mutua esclusione per le regioni critiche */
semaforo s[N];                              /*  un semaforo per ogni filosofo */

void filosofo (int i)                       /* numero del filosofo, da 0 a N-1 */
{
        while (TRUE) {                      /* ripeti per sempre */
                pensa();                    /* il filosofo sta pensando */
                prendi_forchette(i)         /* prende due forchette o si blocca */
                mangia( );                  /* yum-yum, spaghetti */
                riponi_forchette(i);        /* ripone entrambe le forchette sulla tavola */
        }
}

void prendi_forchette(int i)                /* numero del filosofo, da 0 a N-1 */
{
        wait(mutex);                        /* entra nella regione critica */
        stato[i] = AFFAMATO;                /*  il filosofo è affamato */
        test(i);                            /* cerca di prendere 2 forchette */
        signal(mutex);                      /* esce dalla regione critica*/
        wait(s[i])                          /* si blocca se non ha ottenuto le forchette */
        }
}

void riponi_forchette (int i)               /* numero del filosofo, da 0 a N-1 */
{
        wait(mutex);                        /* entra nella regione critica */
        stato[i] = PENSA;                   /* il filosofo ha finito di mangiare */
        test(SINISTRA);                     /* controlla se il vicino di sinistra può mangiare */
        test(DESTRA);                       /* controlla se il vicino di destra può mangiare */
        signal(mutex);                      /* esce dalla regione critica*/
}

void test(int i)                            /* numero del filosofo, da 0 a N-1 */
{
        if (stato[i] ==AFFAMATO && stato[SINISTRA] != MANGIA &&
                                    stato[DESTRA] != MANGIA) {
                stato[i] = MANGIA;
                signal(s[i]);
        }
}
```

Figura 6.15 *Soluzione proposta al problema dei filosofi a tavola*

retta perché si può avere starvation; si trovi un tale controesempio. *Suggerimento*: si consideri il caso di cinque filosofi e si supponga che siano voraci, cioè che pensino poco e vogliano sempre mangiare. Appena un filosofo ha finito di mangiare, quasi subito diventa nuovamente affamato. Quindi si consideri una configurazione in cui due filosofi stiano mangiando e gli altri tre siano bloccati e affamati.

6.15 Si supponga che ci siano due tipi di filosofi: un tipo prende sempre la forchetta di sinistra per prima (un mancino), e l'altro tipo prende la destra. Il comportamento di un mancino è definito nella figura 6.11; quello di un destro è il seguente:

```
begin
    repeat
        pensa;
        wait(forchetta[(i+1) mod 5]);
        wait(forchetta[i]);
        mangia;
        signal(forchetta[i]);
        signal(forchetta[(i+1) mod 5])
    forever
end.
```

Si dimostri quanto segue:

a. Qualunque disposizione della tavola con almeno un destro e un mancino evita lo stallo.

b. Qualunque disposizione della tavola con almeno un destro e un mancino evita la starvation.

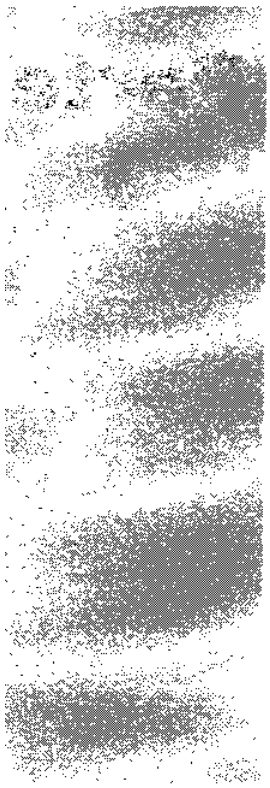
LA MEMORIA

Parte 3

Uno delle parti più difficili nella progettazione di un Sistema Operativo è la gestione della memoria. Sebbene il costo della memoria sia diminuito drasticamente e, di conseguenza, sia aumentata la dimensione della memoria principale sulle macchine moderne, raggiungendo l'ordine di grandezza del gigabyte, la memoria principale non è mai sufficiente per contenere tutti i programmi e le strutture dati dei processi attivi e del sistema operativo. Di conseguenza, un compito centrale del sistema operativo è la gestione della memoria, in altre parole, l'inserimento e la rimozione di blocchi di dati nella memoria secondaria. In ogni modo, l'I/O di porzioni di memoria è un'operazione lenta, e la sua velocità, rispetto al tempo d'esecuzione di un'istruzione del processore, diminuisce sempre più con il passare degli anni. Per tenere il processore, o i processori, occupati e mantenere l'efficienza, il sistema operativo deve, quindi, programmare con attenzione il trasferimento da e per la memoria secondaria, per minimizzare l'effetto dell'I/O di porzioni di memoria sulle prestazioni.

Capitolo 7
La gestione della memoria

Il Capitolo 7 fornisce una panoramica sui meccanismi fondamentali utilizzati per gestire la memoria: sono dapprima presentati i requisiti basilari di qualunque schema per la gestione della memoria, e successivamente il partizionamento della memoria, una tecnica usata solo in casi speciali, come la gestione della memoria del kernel. Tuttavia, un esame del partizionamento della memoria chiarisce molti problemi di progettazione della gestione della memoria. Il resto del capitolo si occupa di due tecniche basilari per la costruzione di tutti i sistemi per la gestione della memoria virtuale: paginazione e segmentazione.



Capitolo 8
La memoria virtuale

La memoria virtuale, basata sull'uso della paginazione oppure sulla combinazione di paginazione e segmentazione, è l'approccio quasi universale alla gestione della memoria sulle macchine contemporanee. La memoria virtuale è uno schema trasparente ai programmi applicativi, e consente ad ogni processo di comportarsi come se avesse a sua disposizione una memoria principale illimitata. Per ottenerla, il sistema operativo crea per ogni processo uno spazio d'indirizzamento virtuale su disco, una parte del quale è trasferita nella memoria principale secondo le necessità; in questo modo, molti processi possono condividere una quantità relativamente piccola di memoria principale. Affinché la memoria virtuale lavori efficacemente, sono necessari meccanismi hardware che eseguano le funzioni fondamentali di paginazione e segmentazione, cioè la traduzione tra indirizzi virtuali e reali. Il Capitolo 8 inizia con una panoramica su questi meccanismi hardware, e prosegue dedicandosi ai problemi di progettazione del sistema operativo per la parte relativa alla memoria virtuale.

CAPITOLO 7

LA GESTIONE DELLA MEMORIA

In un sistema monoprogrammato, la memoria principale è divisa in due parti: una parte per il sistema operativo (monitor residente, kernel) e una parte per il programma che è al momento in esecuzione. In un sistema multiprogrammato, la parte "utente" della memoria deve essere ulteriormente suddivisa per ospitare processi multipli. Tale suddivisione è effettuata dinamicamente dal sistema operativo ed è detta **gestione della memoria**

La gestione della memoria efficiente, in un sistema multiprogrammato, è vitale. Se sono in memoria soltanto pochi processi, allora per la maggior parte del tempo tutti i processi aspetteranno l'I/O, ed il processore non sarà occupato: di conseguenza, la memoria dovrà essere allocata efficientemente per immettervi più processi possibili.

Questo capitolo introdurrà i requisiti principali per una buona gestione della memoria; di seguito, presenterà alcuni semplici schemi che sono stati utilizzati in passato, focalizzando l'attenzione sul fatto che un programma, per essere eseguito, deve essere caricato nella memoria principale. Questa discussione serve ad introdurre alcuni principi fondamentali per la gestione della memoria.

7.1 Requisiti della gestione della memoria

Esaminando i vari meccanismi e le varie politiche associati alla gestione della memoria, è necessario tenere in considerazione i requisiti fondamentali che il gestore della memoria deve

soddisfare. [LIST93] suggerisce cinque argomenti:

- Rilocazione
- Protezione
- Condivisione
- Organizzazione logica
- Organizzazione fisica.

Rilocazione

In un sistema multiprogrammato, la memoria principale disponibile è di solito condivisa tra un certo numero di processi. Generalmente, un programmatore non può sapere in anticipo quali altri programmi risiederanno nella memoria principale al momento dell'esecuzione del suo programma. Inoltre, bisognerebbe essere in grado di trasferire efficientemente i programmi fra disco e memoria principale per massimizzare l'utilizzazione del processore avendo, quindi, a disposizione un grande insieme di processi pronti per essere eseguiti. Una volta che il programma è stato trasferito sul disco, sarebbe abbastanza limitativo imporre che, quando sarà nuovamente trasferito in memoria, debba essere allocato nella stessa regione della memoria principale occupata in precedenza.

Non è perciò possibile sapere in anticipo dove un programma dovrebbe essere allocato, e dobbiamo permettere alle operazioni di trasferimento da e per la memoria secondaria di muovere il programma nella memoria principale. Questo provoca alcuni problemi tecnici in relazione all'indirizzamento, come illustrato nella Figura 7.1. La figura mostra un'immagine di processo; per semplicità, si supponga che essa occupi regioni contigue della memoria principale. Chiaramente, il sistema operativo dovrà conoscere la locazione delle informazioni di controllo del processo e dello stack d'esecuzione, così come il punto d'inizio dell'esecuzione del programma per questo processo. Poiché il sistema operativo gestisce la memoria, ed è anche responsabile del caricamento del processo nella memoria principale, questi indirizzi sono facilmente ottenibili. Il processore dovrà anche occuparsi dei riferimenti alla memoria all'interno del programma: le istruzioni di trasferimento di controllo contengono l'indirizzo dell'istruzione che deve essere eseguita al passo successivo, e le istruzioni di riferimento ai dati contengono l'indirizzo di un byte o di una parola. In qualche modo, l'hardware del processore ed il software del sistema operativo devono essere in grado di tradurre i riferimenti alla memoria trovati nel codice del programma in indirizzi fisici effettivi di memoria, riflettendo l'allocazione corrente del programma nella memoria principale.

Protezione

Ogni processo deve essere protetto contro interferenze indesiderate da parte di altri processi, siano esse accidentali o intenzionali; perciò, i programmi degli altri processi non dovrebbero avere riferimenti a locazioni di memoria di un processo, allo scopo di leggervi o scrivervi senza

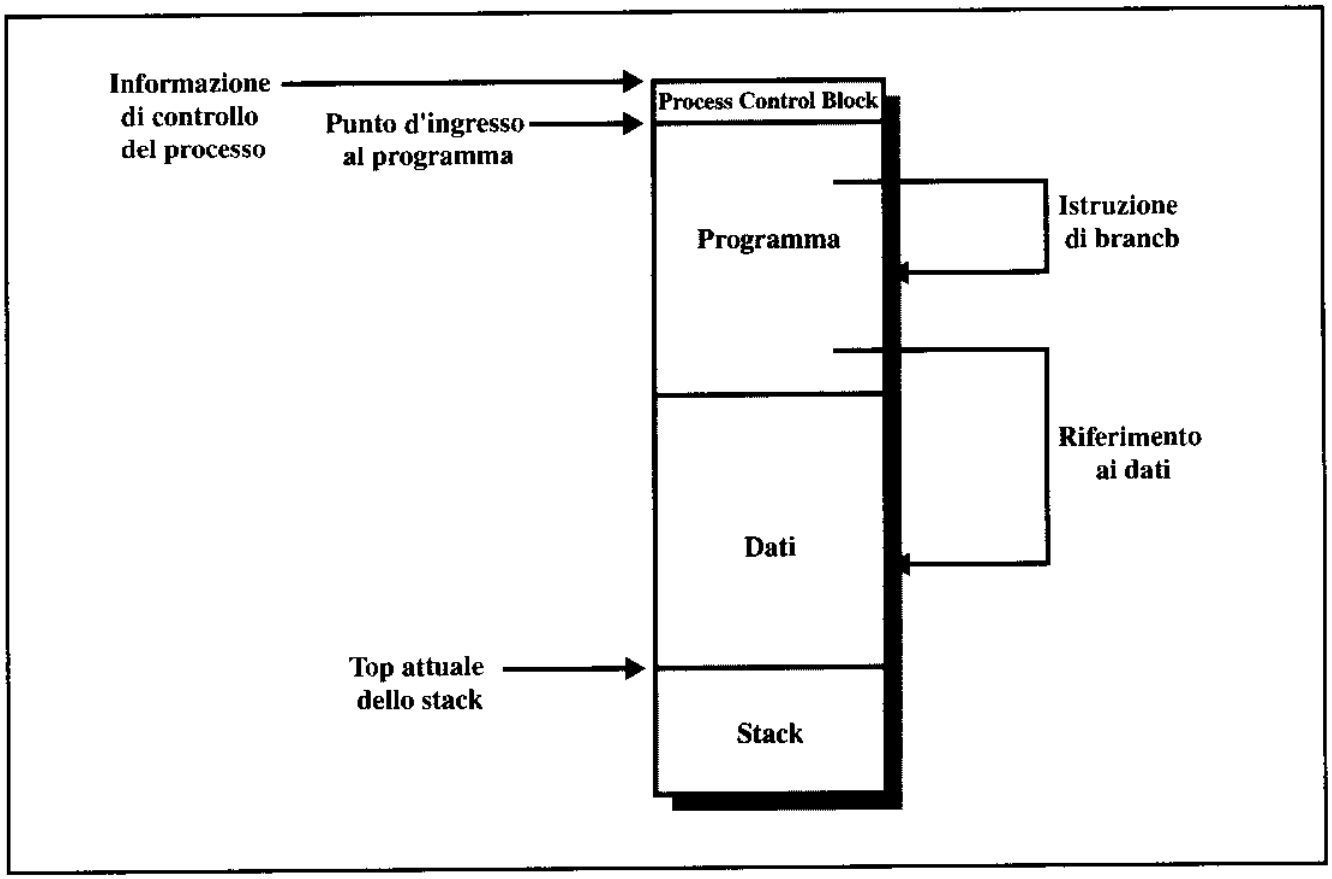


Figura 7.1 *Requisiti per l'indirizzamento di un processo*

permesso: il requisito di rilocazione sembra che aumenti la difficoltà di soddisfare il requisito di protezione. Poiché la locazione di un programma nella memoria principale è sconosciuta, è impossibile controllare gli indirizzi assoluti a tempo di compilazione per assicurare la protezione. Inoltre, la maggioranza dei linguaggi di programmazione permette il calcolo dinamico degli indirizzi a tempo di esecuzione (per esempio, calcolando un indice di array o un puntatore in una struttura dati). Quindi tutti i riferimenti alla memoria generati da un processo devono essere controllati a tempo di esecuzione per assicurarsi che si riferiscano solo allo spazio di memoria allocato per quel processo. Fortunatamente, il meccanismo che supporta la rilocazione soddisfa anche il requisito di protezione.

L'immagine del processo in Figura 7.1 illustra il requisito di protezione. Normalmente, un processo utente non può accedere a nessuna parte del sistema operativo, sia esso un programma o dati. Inoltre, solitamente un programma appartenente ad un processo non può trasferire il controllo a un'istruzione di un altro processo. Se non si formulano particolari ipotesi, un programma in un processo non può accedere all'area dati di un altro processo; quindi, il processore deve essere in grado di bloccare tali istruzioni al momento dell'esecuzione.

Bisogna notare che, sulla base degli esempi proposti, il requisito di protezione della memoria deve essere soddisfatto dal processore (hardware) piuttosto che dal sistema operativo (software). Questo perché il sistema operativo non può conoscere in anticipo tutti i riferimenti di memoria che un programma farà. Anche se tale anticipazione fosse possibile, sarebbe troppo costoso esaminare in anticipo ogni programma per cercare possibili violazioni di riferimenti alla memoria. Perciò, si può valutare solo la possibilità di riferimento alla memoria (accesso a dati o trasfe-

rimento di controllo) al momento dell'esecuzione dell'istruzione che fa il riferimento. Per realizzare questo, l'hardware del processore deve avere opportune capacità.

Condivisione

Qualunque meccanismo di protezione deve permettere a diversi processi di accedere alla stessa porzione di memoria principale. Per esempio, se un certo numero di processi sta eseguendo lo stesso programma, è vantaggioso permettere a tutti i processi di accedere alla stessa copia del programma, piuttosto che darne a ciascuno una copia separata. I processi che cooperano ad uno stesso compito possono avere bisogno di condividere l'accesso alla stessa struttura dati; quindi, il sistema per la gestione della memoria deve permettere l'accesso controllato ad aree condivise di memoria senza comprometterne la protezione. In seguito, si vedrà che il meccanismo usato per supportare la rilocazione supporta anche la capacità di condivisione.

Organizzazione logica

Quasi invariabilmente, la memoria principale in un sistema di elaborazione è organizzata come uno spazio di indirizzamento lineare, o monodimensionale, composto da una sequenza di byte o parole. La memoria secondaria, a livello fisico, è organizzata in modo simile. Mentre questa organizzazione rispecchia fedelmente l'hardware dei giorni nostri, non corrisponde al modo in cui tipicamente si costruiscono i programmi. La maggior parte dei programmi è organizzata in moduli, alcuni dei quali non sono modificabili - **read only** (solo leggibili), **execute only** (solo eseguibili) -, mentre altri contengono dati che possono essere modificati. Se il sistema operativo e l'hardware del computer possono efficientemente occuparsi dei programmi utente e dei dati, considerandolo come moduli, allora si può ottenere un certo numero di vantaggi:

1. I moduli possono essere scritti e compilati indipendentemente, con tutti i riferimenti da un modulo all'altro risolti dal sistema a tempo di esecuzione.
2. Con un piccolo ulteriore sovraccarico, è possibile dare a moduli diversi, gradi di protezione diversi (read only, execute only).
3. È possibile introdurre meccanismi tramite i quali i moduli possono essere condivisi tra processi. Fornire la condivisione a livello di moduli corrisponde a vedere il problema dalla parte dell'utente, ed è quindi facile per l'utente specificare la condivisione desiderata.

Lo strumento che soddisfa più rapidamente queste richieste è la segmentazione, che è una delle tecniche di gestione della memoria esaminate in questo capitolo.

Organizzazione fisica

Come è già stato discusso nella Sezione 1.5, la memoria del computer è organizzata in almeno due livelli: la memoria principale e la memoria secondaria. La memoria principale permette un accesso rapido ma ad un costo relativamente alto ed inoltre, è volatile; cioè, non fornisce una memorizzazione permanente. La memoria secondaria è, invece, più lenta ma meno costosa della memoria principale e solitamente non è volatile; perciò, una memoria secondaria di grande

capacità può essere utilizzata per la memorizzazione a lungo termine di programmi e dati, mentre una piccola memoria principale conserva i programmi e i dati correntemente in uso.

In questo schema a due livelli, l'organizzazione del flusso di informazione tra la memoria principale e la memoria secondaria è un compito fondamentale del sistema. La responsabilità di questo flusso potrebbe essere assegnata al singolo programmatore, ma questo non è pratico né desiderabile per due ragioni:

1. La memoria principale disponibile per un programma e i suoi dati potrebbe essere insufficiente. In questo caso, il programmatore dovrebbe adottare una tecnica detta **overlay**, in cui il programma e i dati sono organizzati in modo tale che ai vari moduli si possa assegnare la stessa regione di memoria, con un programma principale responsabile del trasferimento dei moduli da e per la memoria secondaria, a seconda della necessità. Anche con l'aiuto di un compilatore, la programmazione di un overlay fa perdere tempo al programmatore.
2. In un ambiente in multiprogrammazione, il programmatore non conosce, al momento della scrittura del codice, quanto spazio sarà disponibile o dove sarà questo spazio.

È chiaro, poi, che l'operazione di spostamento dell'informazione tra due livelli di memoria, l'essenza del gestore della memoria, deve essere un compito del sistema.

7.2 Il partizionamento della memoria

Il compito principale del gestore della memoria è portare i programmi nella memoria principale perché il processore li esegua. In quasi tutti i moderni sistemi in multiprogrammazione, questo coinvolge uno schema raffinato conosciuto come memoria virtuale. La memoria virtuale è, a sua volta, basata sull'uso di una o entrambe le tecniche di base: segmentazione e paginazione. Prima di dare un'occhiata a queste tecniche di memoria virtuale, è opportuno prendere confidenza con le tecniche più semplici, che non la coinvolgono (Tabella 7.1). Una di queste tecniche, il partizionamento, è stata usata con diverse variazioni, in alcuni sistemi operativi ora obsoleti. Le altre due tecniche, la paginazione semplice e la segmentazione semplice, non sono usate da sole, ma il discorso sulla memoria virtuale diverrà più chiaro se si comprendono queste due tecniche senza abbinarle a considerazioni sulla memoria virtuale.

Partizionamento fisso

Nella maggior parte degli schemi per la gestione della memoria, si può supporre che il sistema operativo occupi una porzione fissa della memoria principale e che il resto della memoria principale sia disponibile per essere utilizzata da molti processi. Lo schema più semplice per la gestione di questa memoria disponibile è il suo partizionamento in regioni con confini fissati.

Dimensioni della partizione

La Figura 7.2 mostra due alternative per il partizionamento fisso. Una possibilità è quella di fare uso di partizioni di uguali dimensioni: in questo caso, qualunque processo la cui dimensio-

Tabella 7.1 *Tecniche per la grstione della memoria*

Tecnica	Descrizione	Vantaggi	Svantaggi
Partizionamento fisso	La memoria principale è divisa in un certo numero di partizioni statiche al momento della creazione del sistema. Un processo può essere caricato in una partizione di dimensione maggiore o uguale.	Semplice da implementare; poco overhead del sistema operativo	Uso inefficiente della memoria a causa della frammentazione interna; il numero dei processi attivi è fisso.
Partizionamento dinamico	Le partizioni sono create dinamicamente, quindi ogni processo è caricato in una partizione che ha esattamente la stessa dimensione del processo	Non presenta frammentazione interna; uso più efficiente della memoria principale	Uso inefficiente del processore perché è necessario compattare per opporsi alla frammentazione esterna
Paginazione semplice	La memoria principale è suddivisa in frame di uguale dimensione. Ogni processo è diviso in pagine di uguale dimensione, aventi la stessa lunghezza dei frame. Un processo è caricato caricando tutte le sue pagine nei frame disponibili, non necessariamente contigui.	Non presenta frammentazione esterna	Poca frammentazione interna
Segmentazione semplice	Ogni processo è diviso in segmenti. Si carica un processo caricando tutti i suoi segmenti in partizioni dinamiche, non necessariamente contigue.	Non presenta frammentazione interna	Migliora l'uso della memoria e riduce l'overhead rispetto al partizionamento dinamico.
Paginazione con memoria virtuale	Come la paginazione semplice, ma non è necessario caricare tutte le pagine di un processo. Le pagine necessarie non presenti in memoria sono caricate successivamente in modo automatico.	Non presenta frammentazione esterna; livello più alto di multiprogrammazione; grande spazio virtuale per il processo.	Overhead per la complessità di gestione della memoria
Segmentazione con memoria virtuale	Come la segmentazione semplice, ma non è necessario caricare tutti i segmenti del processo. I segmenti necessari non presenti in memoria sono caricati successivamente in modo automatico.	Non presenta frammentazione interna, livello di multiprogrammazione più alto; grande spazio virtuale per il processo; supporto per la protezione e la condivisione	Overhead per la complessità di gestione della memoria

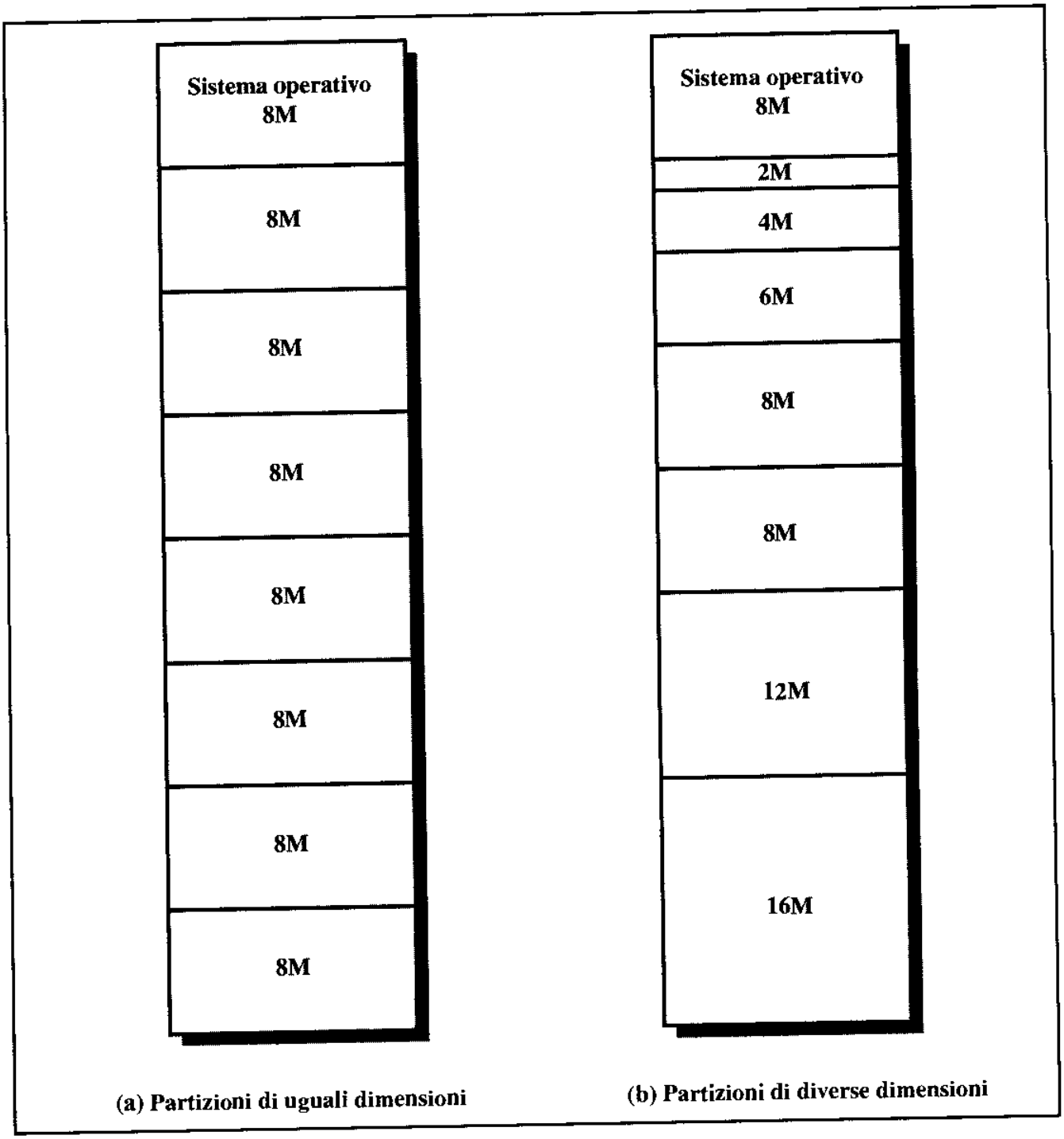


Figura 7.2 *Esempi di partizionamento fisso di una memoria da 64MB*

ne è minore o uguale alla dimensione della partizione può essere caricato in qualunque partizione disponibile. Se tutte le partizioni sono piene e nessun processo è nello stato Ready o nello stato Running, il sistema operativo può trasferire un processo fuori da una qualunque partizione e caricarvici un altro processo, in modo che ci sia lavoro per il processore.

Sono due le difficoltà che si presentano usando partizioni fisse della stessa dimensione:

- Un programma può essere troppo grande per risiedere in una partizione. In questo caso, il programmatore deve progettare il programma con l'uso di overlay, in modo da averne solo

una porzione residente nella memoria principale in ogni momento. Quando c'è bisogno di un modulo non presente in memoria, il programma dell'utente deve caricare quel modulo nella partizione dedicata al programma, sovrapponendolo ai programmi o dati già presenti.

- L'utilizzazione della memoria principale è estremamente inefficiente: infatti, qualunque programma, indipendentemente dalla sua dimensione, occupa un'intera partizione. Nell'esempio riportato, ci può essere un programma la cui lunghezza è minore di 2MB, tuttavia esso occupa una partizione di 8MB ogni volta che è trasferito in memoria. Questo fenomeno (**frammentazione interna**) è uno spreco di spazio all'interno di una partizione, poiché il blocco di dati caricato ha dimensioni minori rispetto a quelle della partizione.

Entrambi questi problemi possono essere minimizzati, sebbene non risolti, usando partizioni con diversa dimensione (Figura 7.2b). In questo esempio, programmi fino a 16 megabyte possono essere allocati senza overlay. Le partizioni più piccole di 8MB accoglieranno i programmi più piccoli, con meno frammentazione interna.

Algoritmo di allocazione

Con partizioni di uguale dimensione, l'allocazione dei processi in memoria è banale: finché c'è una partizione disponibile, un processo può esservi caricato. Poiché le partizioni hanno tutte la stessa dimensione, non ha importanza quale partizione è usata; e se tutte la partizioni fossero occupate da processi che non sono Ready, uno di questi processi deve essere trasferito per far posto ad un nuovo processo. Lo schedulatore, argomento della Parte 4, deciderà quale processo trasferire.

Utilizzando partizioni di diversa dimensione, è possibile scegliere tra due metodi di assegnazione dei processi alle partizioni. Il metodo più semplice consiste nell'assegnare ogni processo alla partizione più piccola che lo contenga[1]. In questo caso, è necessaria una coda di schedulazione per ogni partizione, per mantenere i processi trasferiti all'esterno destinati a quella partizione (Figura 7.3a). Il vantaggio di questo approccio sta nel fatto che i processi sono sempre allocati in modo da minimizzare la frammentazione all'interno di una partizione.

Sebbene questa tecnica sembri ottima dal punto di vista di una singola partizione, non è ottima dal punto di vista del sistema nel suo insieme. In Figura 7.2b, per esempio, si consideri il caso in cui, ad un certo punto, non ci siano processi con una dimensione compresa tra i 12 e i 16 megabyte. In questo caso, la partizione di 16MB rimarrà inutilizzata, anche se alcuni processi più piccoli potrebbero esservi contenuti. Di conseguenza, è preferibile impiegare una sola coda per tutti i processi (Figura 7.3b) e, al momento di caricare un processo nella memoria principale, selezionare la partizione disponibile più piccola che lo contenga; ma se tutte le partizioni sono occupate, allora si deve scaricare un processo sulla memoria secondaria. Si preferisce quindi scaricare la partizione più piccola che contenga il processo in arrivo, ma è anche possibile considerare altri fattori, come la priorità, o dare preferenza al trasferimento di processi bloccati piuttosto che processi in stato Ready.

[1] Questo implica che sia noto il massimo quantitativo di memoria che un processo richiederà, il che non è sempre vero. Se non si sa quanto può diventare grosso un processo, le uniche alternative sono uno schema di overlay o l'uso della memoria virtuale.

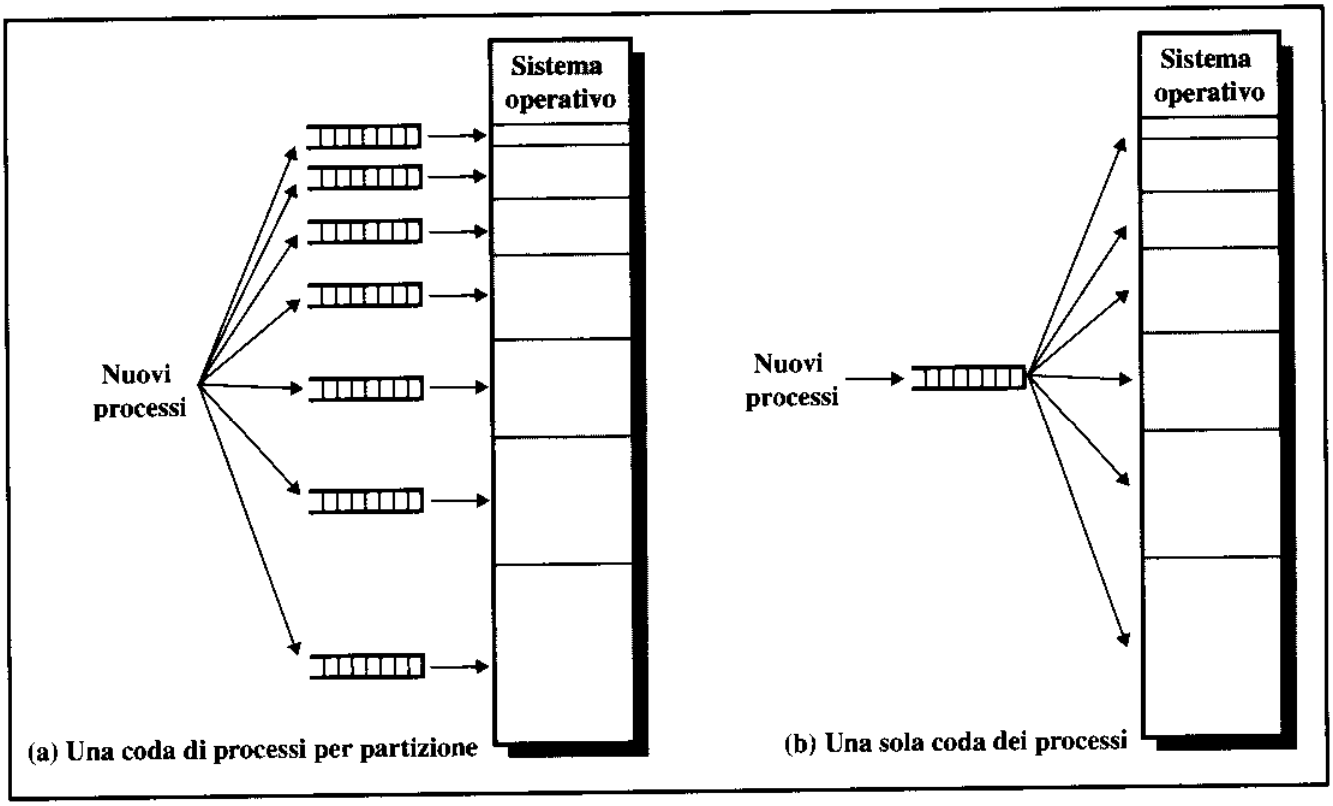


Figura 7.3 *Allocazione della memoria per partizionamento fisso*

L'uso di partizioni con diversa dimensione fornisce una certa flessibilità al partizionamento fisso; inoltre, si può affermare che gli schemi per il partizionamento fisso sono relativamente semplici e richiedono un sovraccarico minimo al software del sistema operativo. Nonostante ciò, la tecnica presenta alcuni svantaggi:

- Il numero di partizioni specificato al tempo della generazione del sistema limita il numero di processi attivi (non sospesi) nel sistema.
- Poiché le dimensioni della partizione sono predefinite al momento della generazione del sistema, i piccoli job non utilizzeranno efficientemente lo spazio della partizione. In un ambiente dove le richieste di memoria di tutti i job sono conosciute in anticipo, questo può essere ragionevole, ma nella maggioranza dei casi, è una tecnica inefficiente.

Oggi, il partizionamento fisso è praticamente sconosciuto. Un esempio di un famoso sistema operativo che usava questa tecnica era il sistema operativo per mainframe dell'IBM, OS/MFT (Multiprogramming with a fixed number of tasks - Multiprogrammazione con un numero fissato di task).

Partizionamento dinamico

Per superare alcune delle difficoltà che si incontrano nel partizionamento fisso, è stato sviluppato un approccio conosciuto come partizionamento dinamico. Anche questo approccio è stato soppiantato da tecniche di gestione della memoria più sofisticate; un celebre sistema operativo che ha fatto uso di questa tecnica è stato l'OS/MVT, il sistema operativo per mainframe dell'IBM.

Con il partizionamento dinamico, si usano partizioni in numero e lunghezza variabile; quando un processo è caricato nella memoria principale, gli è allocata tanta memoria quanta richiesta e mai di più. Un esempio, in cui si usa 1MB di memoria principale, è mostrato in figura 7.4: inizialmente, la memoria principale è vuota, eccetto che per il sistema operativo (a). In seguito, i primi tre processi sono caricati, a cominciare da dove il sistema operativo finisce, ed occupando solo lo spazio necessario per ogni processo (b, c, d): facendo questo, si lascia un "buco" alla fine della memoria, che è troppo piccolo per un quarto processo. Ad un certo punto, nessuno dei processi è Ready: allora, il sistema operativo trasferisce su disco il processo 2 (e), che lascia spazio sufficiente a caricare un nuovo processo, il processo 4 (f), ma poiché il processo 4 è più piccolo del processo 2, si crea un altro piccolo buco. In seguito, succede nuovamente che nessuno dei processi nella memoria principale sia Ready, mentre il processo 2, nello stato Ready-Suspend, sarebbe disponibile. Poiché in memoria non c'è spazio sufficiente per il processo 2, il sistema operativo trasferisce su disco il processo 1 (g) e porta nuovamente in memoria il processo 2 (h).

Come mostra l'esempio, questo metodo comincia bene, ma prima o poi, porta ad avere tanti piccoli buchi in memoria; con il passare del tempo, la memoria diventa sempre più frammentata e l'utilizzo della memoria peggiora. Questo fenomeno è chiamato **frammentazione esterna**, e si riferisce al fatto che la memoria esterna a tutte le partizioni diventa sempre più frammentata: si confronti questo fenomeno con la frammentazione interna, di cui si è discusso precedentemente.

Una tecnica per superare la **frammentazione esterna** è la **compattazione**: periodicamente, il sistema operativo sposta i processi in modo tale che essi siano contigui e tutta la memoria libera sia riunita in un blocco. Per esempio, nella Figura 7.4h, la compattazione restituirà un blocco di memoria libera di lunghezza 256KB, che può essere sufficiente per caricare un ulteriore processo. Il difetto della compattazione deriva dal suo elevato costo computazionale, che fa perdere tempo al processore. La compattazione implica, inoltre, una capacità di rilocazione dinamica, cioè, deve essere possibile spostare un programma da una regione all'altra della memoria principale senza invalidarne i riferimenti alla memoria.

Algoritmo di allocazione

Poiché la compattazione di memoria è costosa in termini di tempo, il progettista del sistema operativo deve decidere come assegnare i processi alla memoria (come tappare i buchi) in modo intelligente. Quando è il momento di caricare o trasferire un processo nella memoria principale, se c'è più di un blocco di memoria libero di dimensione sufficiente, il sistema operativo deve decidere quale blocco di memoria allocare.

Si possono considerare tre algoritmi di allocazione: **best-fit**, **first-fit** e **next-fit**, tutti, naturalmente, si limitano a scegliere tra i blocchi liberi di memoria principale che sono uguali, o più grandi, del processo che deve essere immesso in memoria. Il best-fit sceglie il blocco che è più vicino in dimensione alla richiesta. Il first-fit scandisce la memoria dall'inizio e sceglie il primo blocco disponibile sufficientemente grande. Il next-fit, invece, scandisce la memoria partendo dalla locazione dell'ultima allocazione e sceglie il blocco disponibile successivo, purché sufficientemente grande.

La Figura 7.5a mostra un esempio di configurazione di memoria dopo un certo numero di

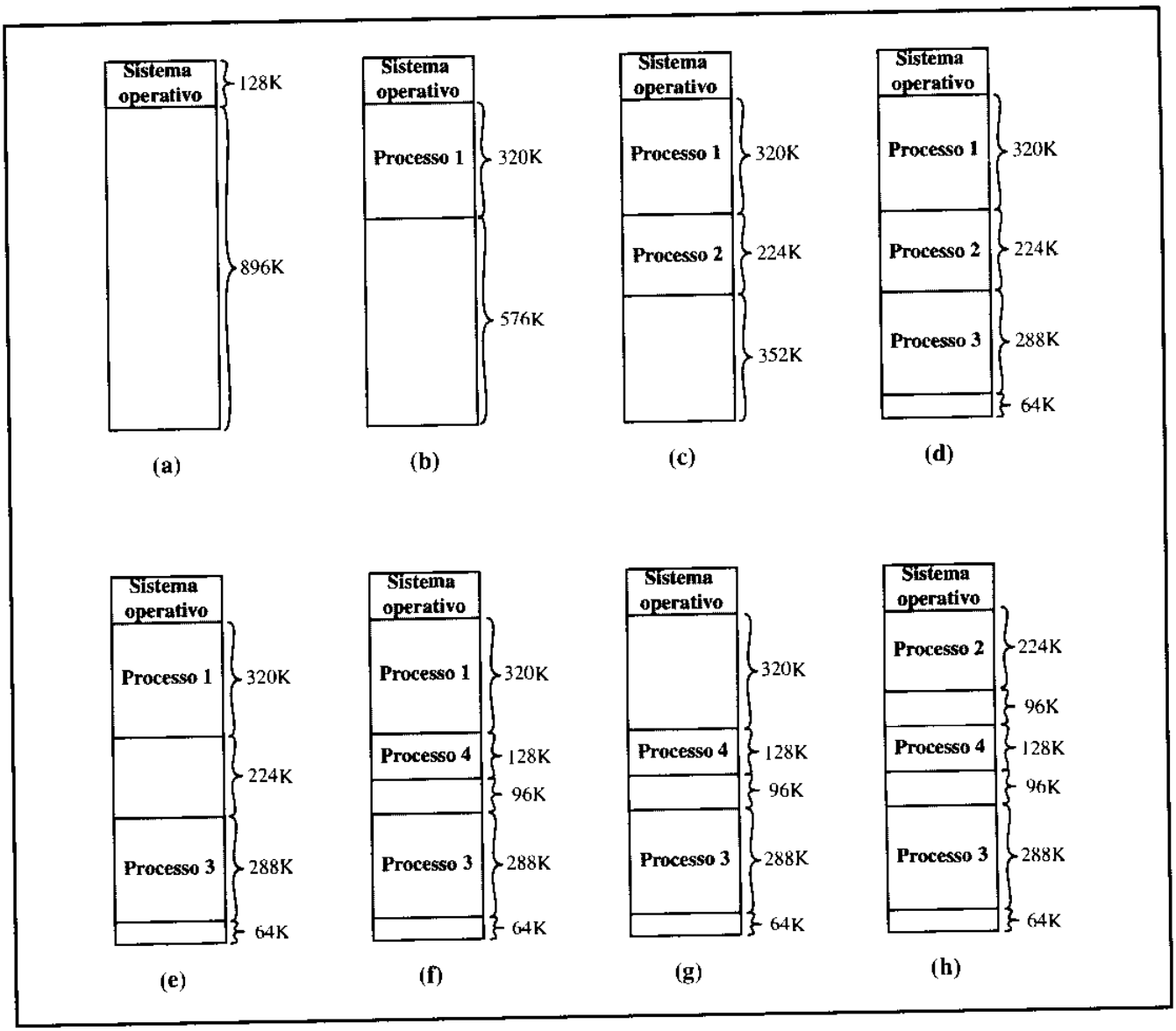


Figura 7.4 *Effetti del partizionamento dinamico*

operazioni di allocazione e trasferimento su disco. L'ultimo blocco usato era un blocco di 22 kilobyte dal quale era stata creata una partizione di 14KB. La Figura 7.5b mostra la differenza tra i tre algoritmi di allocazione nel soddisfare una richiesta di allocazione di 16KB. Il Best-fit cercherà sull'intera lista dei blocchi disponibili, scegliendo il blocco di 18 kilobyte, lasciando un frammento di 2KB. Il First-fit lascerà un frammento di 6KB e il next-fit lascerà un frammento di 20KB.

Quale di questi approcci sia il migliore dipenderà dall'esatta sequenza dei trasferimenti di processi che avverranno e dalla dimensione di quei processi. (Vedere anche [BREN89], [SHOR75] e [BAYS77]). L'algoritmo di first-fit non è solo il più semplice ma solitamente anche il migliore e il più veloce, mentre l'algoritmo di next-fit tende a produrre risultati leggermente peggiori del first-fit: infatti, esso porterà più frequentemente ad allocare entro il blocco libero alla fine della memoria. Il risultato sarà che il più grande blocco di memoria libera, che solitamente compare alla fine dello spazio di memoria, è rapidamente smantellato in piccoli frammenti, perciò la compattazione è richiesta più frequentemente. D'altra parte, l'algoritmo di first-fit tende a riem-

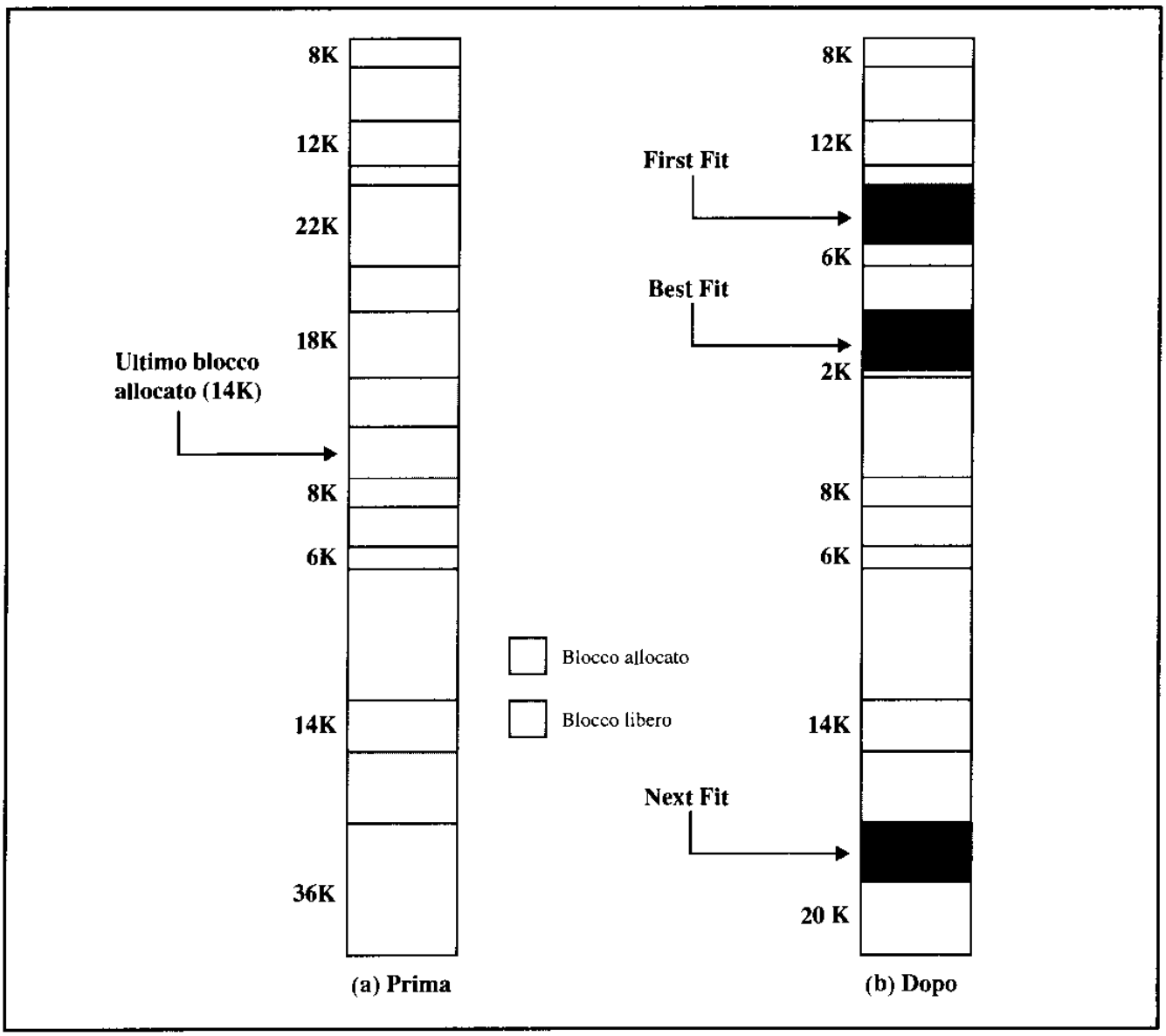


Figura 7.5 *Esempio di configurazione della memoria prima e dopo l'allocazione di un blocco da 16KB*

pire la parte iniziale della memoria di piccole partizioni libere che devono essere esaminate ad ogni passo dell'algoritmo. L'algoritmo di best-fit, nonostante il suo nome, è solitamente il peggiore: poiché esso cerca il blocco più piccolo soddisfacente la richiesta, garantisce che il frammento lasciato è il più piccolo possibile, e siccome ogni richiesta di memoria ne spreca sempre un piccolissimo quantitativo, il risultato è che la memoria principale è rapidamente inframmezzata di blocchi troppo piccoli per soddisfare le richieste di allocazione di memoria. Perciò la compattazione di memoria deve essere fatta più frequentemente che con gli altri algoritmi.

Algoritmo di riallocazione

In un sistema multiprogrammato che usa il partizionamento dinamico, ci sarà un momento in cui tutti i processi residenti nella memoria principale saranno bloccati e la memoria sarà insufficiente per un ulteriore processo, anche dopo una compattazione. Per evitare uno spreco di tempo

di processore, in attesa che un processo attivo si sblocchi, il sistema operativo trasferirà fuori dalla memoria uno dei processi per fare spazio ad un nuovo processo o ad un processo in stato Ready-Suspend, perciò il sistema operativo deve scegliere quale processo rimpiazzare. Poiché l'argomento della riallocazione sarà ampiamente trattato nella parte dedicata ai vari schemi di memoria virtuale, rimandiamo ogni discussione ai prossimi paragrafi.

Buddy System

Entrambi gli schemi di partizionamento dinamico e fisso hanno degli inconvenienti. Uno schema di partizionamento fisso limita il numero di processi attivi, e può usare inefficientemente lo spazio se c'è poca corrispondenza tra la dimensione della partizione disponibile e la dimensione del processo; d'altra parte, uno schema di partizionamento dinamico è più complicato da mantenere e necessita del costo aggiuntivo della compattazione. Un interessante compromesso è il buddy system, o sistema dei compagni ([KNUT97], [PETE77]).

In un buddy system, i blocchi di memoria disponibile hanno dimensione 2^k, con $L \leq K \leq U$, dove

2^L = blocco allocato di dimensione più piccola

2^U = blocco allocato di dimensione più grande; generalmente 2^U è la dimensione dell'intera memoria disponibile per l'allocazione

Inizialmente, l'intero spazio disponibile per l'allocazione è trattato come un singolo blocco di dimensione 2^U. Se è fatta una richiesta di dimensione s tale che $2^{U-1} < s \leq 2^U$, allora è allocato l'intero blocco; altrimenti il blocco è diviso in due "compagni" uguali di dimensione 2^{U-1}. Se $2^{U-2} < s \leq 2^{U-1}$, allora la richiesta è allocata in uno dei due compagni; altrimenti, uno dei due compagni è nuovamente diviso a metà. Questo procedimento continua fino a che non è generato un blocco, più grande o uguale a s, che è allocato per la richiesta di s. In ogni momento il buddy system mantiene una lista di buchi (blocchi non allocati) per ciascuna dimensione 2^i. Un buco può essere rimosso dalla lista $i+1$-esima dividendolo a metà, per creare due compagni di dimensione 2^i da inserire nella lista i-esima. Ogni volta che viene deallocata una coppia di compagni nella lista i-esima, essi sono rimossi da quella lista e i compagni sono riuniti in un singolo blocco nella lista $i+1$-esima. In presenza di una richiesta di allocazione di dimensione k tale che $2^{i-1} < k \leq 2^i$, si usa il seguente algoritmo ricorsivo (da [LIST93]) per trovare un blocco libero di dimensione 2^i:

```
procedure prendibuco (i);
begin
   if i = U + 1 then fallimento;
   if i_list vuota then begin
      prendibuco (i + 1);
      dividi il buco in due compagni;
      metti i compagni nella lista i_list
   end;
   prendi il primo buco in i_list
end;
```

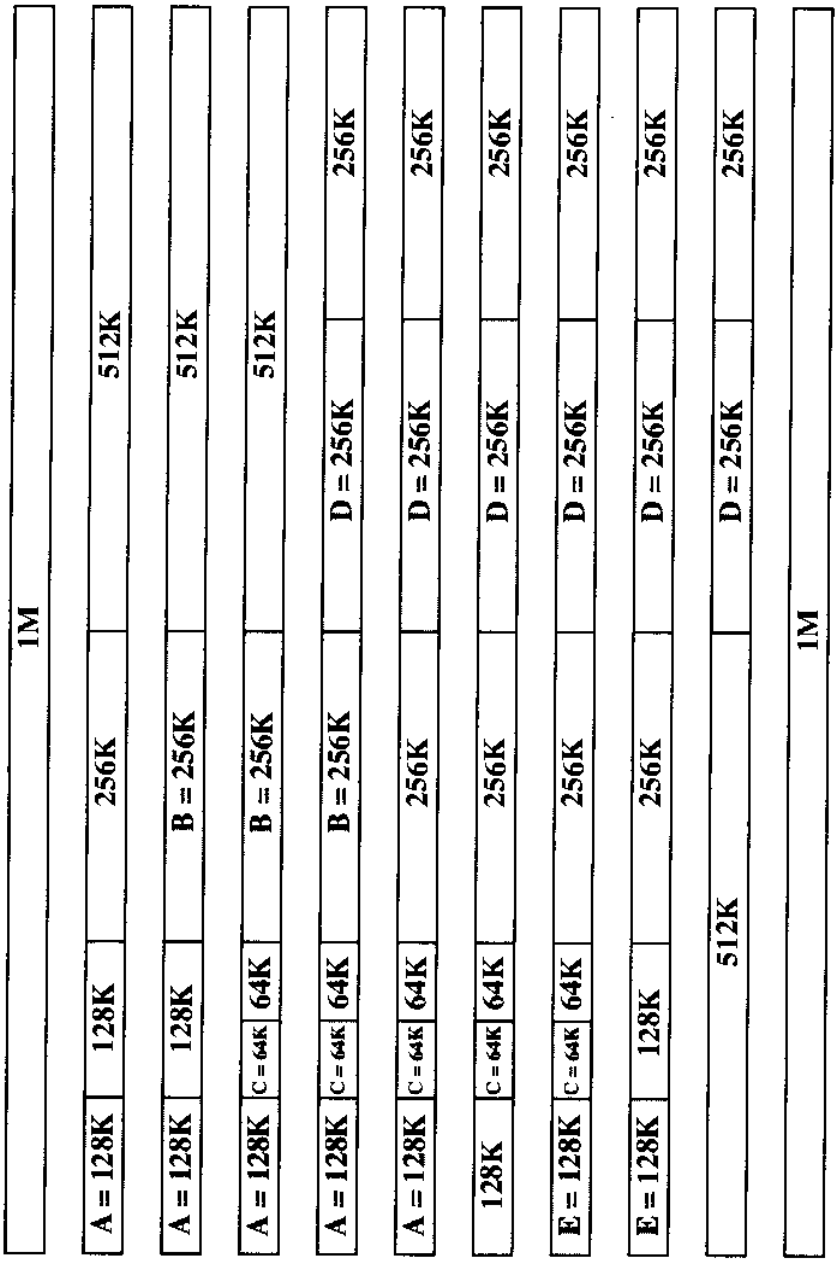


Figura 7.6 *Esempio di buddy system*

La figura 7.6 mostra un esempio usando un blocco iniziale di 1 megabyte. La prima richiesta, A, è di 100KB, per la quale è necessario un blocco di 128KB: il blocco iniziale è diviso in due compagni da 512KB, il primo dei quali è diviso in due compagni da 256KB, e il primo di questi è diviso in due compagni da 128KB, uno di questi ultimi è allocato per A. La richiesta successiva, B, richiede un blocco da 256KB. Un tale blocco è già disponibile ed è allocato. Il procedimento continua, con divisioni e riunificazioni quando e se necessarie; si noti che al rilascio di E, due compagni da 128KB sono riuniti in un blocco da 256KB, che è immediatamente riunito con il suo compagno.

La figura 7.7 mostra una rappresentazione ad albero binario della allocazione di compagni dopo la richiesta di rilascio di B; i nodi foglia rappresentano il partizionamento di memoria corrente. Se due compagni sono nodi foglia, allora almeno uno deve essere allocato; altrimenti essi dovrebbero essere riuniti in un blocco più grande.

Il buddy system è un ragionevole compromesso che supera gli svantaggi degli schemi di partizionamento fisso e variabile, ma nei sistemi operativi di oggi, la memoria virtuale basata sulla paginazione e sulla segmentazione è superiore. Comunque, il buddy system ha trovato applicazione nei sistemi paralleli come un mezzo efficiente per l'allocazione e il rilascio dei programmi eseguiti in parallelo (vedere ad esempio [JOHN92]). Una forma modificata del buddy system è usata per l'allocazione di memoria del kernel di UNIX (descritta nel Capitolo 8).

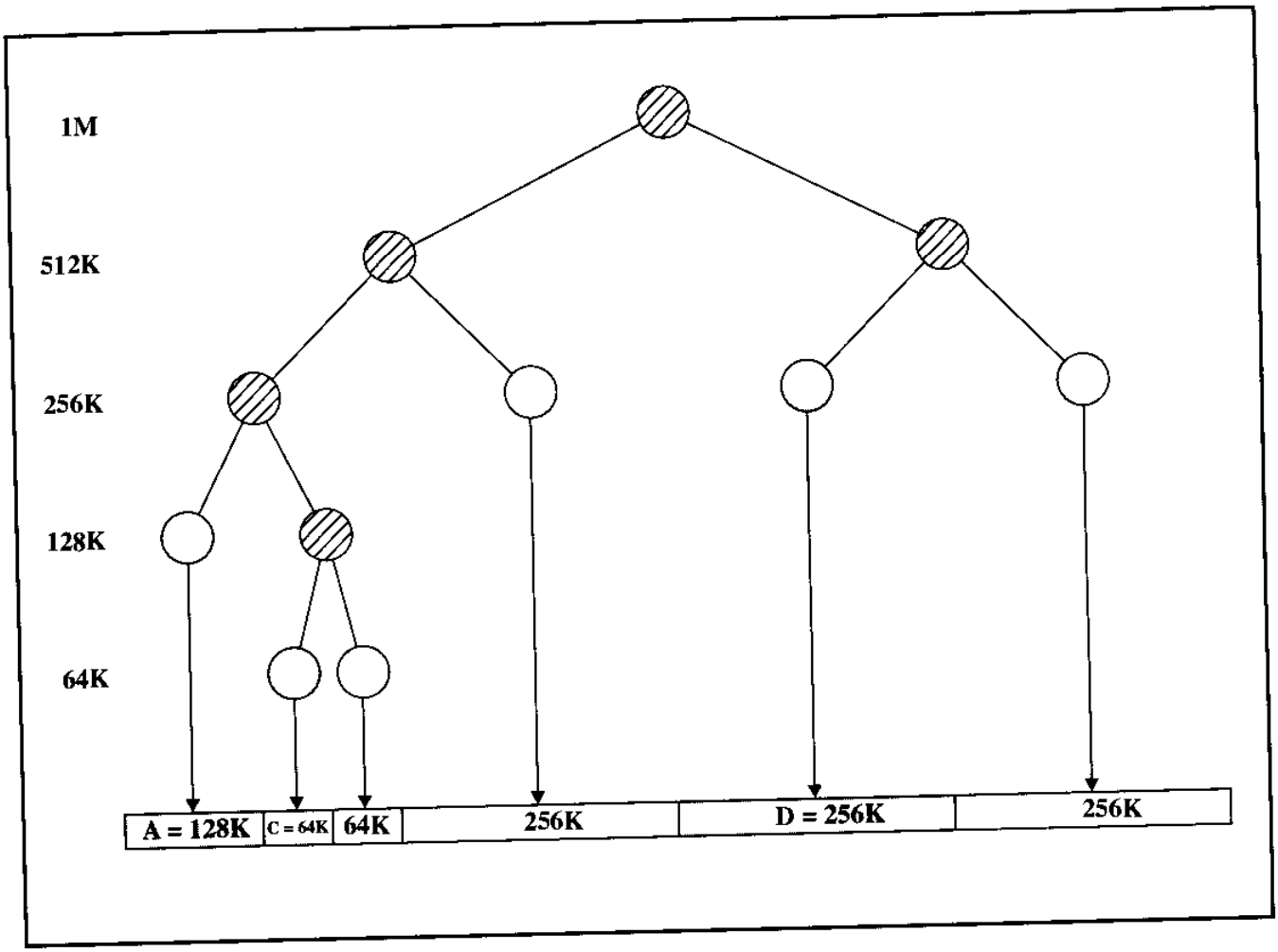


Figura 7.7 *Rappresentazione ad albero di un buddy system*

Rilocazione

Prima di considerare come trattare i problemi del partizionamento, bisogna chiarire un punto rimasto in sospeso, circa l'allocazione dei processi in memoria. Quando si usa lo schema di partizionamento fisso di Figura 7.3a, si può ritenere che un processo sia sempre assegnato alla stessa partizione, cioè che la partizione selezionata al momento del caricamento di un nuovo processo sarà la stessa in cui allocarlo ogni volta che rientrerà in memoria. In questo caso, si può usare un semplice programma di caricamento con rilocazione, come quello descritto nell'Appendice 7A. Quando un processo è caricato per la prima volta, tutti gli indirizzi relativi in memoria, presenti nel codice, sono sostituiti dagli indirizzi assoluti della memoria principale, determinati dall'indirizzo base del processo caricato.

Nel caso di partizioni di uguale dimensione e nel caso di una sola coda dei processi per partizioni diverse, un processo può occupare partizioni diverse nel corso della sua vita. Quando un'immagine di processo è creata per la prima volta, è caricata in una partizione della memoria principale; in seguito, il processo può essere trasferito fuori dalla memoria, ma quando è successivamente reinserito, può essere assegnato ad una partizione diversa dalla precedente; lo stesso può succedere con il partizionamento dinamico. Si osservi nelle Figure 7.4c e 7.4h che il processo 2 occupa due diverse regioni di memoria nelle due occasioni in cui è reimmesso in memoria; inoltre, quando si usa la compattazione, i processi sono spostati mentre sono nella memoria principale. Perciò, le locazioni (delle istruzioni e dei dati) cui un processo fa riferimento non sono fisse, ma cambieranno ogni volta che un processo è caricato in memoria, o spostato. Per risolvere questo problema, si distinguono diversi tipi di indirizzo: un **indirizzo logico** è un riferimento a una locazione di memoria, indipendente dall'assegnazione attuale dei dati in memoria: deve essere tradotto in indirizzo fisico prima di ottenere l'accesso alla memoria. Un **indirizzo relativo** è un caso particolare di indirizzo logico, in cui l'indirizzo è espresso come una locazione relativa ad un punto conosciuto, solitamente l'inizio del programma. Un **indirizzo fisico**, o indirizzo assoluto, è una locazione effettiva nella memoria principale.

I programmi che utilizzano gli indirizzi relativi in memoria sono caricati usando il caricamento dinamico a tempo di esecuzione (per dettagli vedere l'appendice 7A). Questo significa che tutti i riferimenti a memoria, nel processo caricato, sono relativi all'origine del programma. Perciò, è necessario un meccanismo hardware per tradurre gli indirizzi relativi in indirizzi fisici nella memoria principale al tempo di esecuzione dell'istruzione che contiene il riferimento.

La Figura 7.8 mostra il modo tipico in cui si compie questa traduzione di indirizzo: quando un processo è in stato Running, un registro speciale del processore, qualche volta chiamato **registro base**, è caricato con l'indirizzo iniziale del processo nella memoria principale. C'è anche un **registro limite,** che indica la locazione finale del programma; questi valori devono essere impostati quando il programma è caricato in memoria, o quando l'immagine del processo è trasferita dal disco. Durante l'esecuzione del processo, si incontrano indirizzi relativi: tra essi vi saranno il contenuto del registro di istruzione, gli indirizzi delle istruzioni che si incontrano nelle istruzioni di trasferimento di controllo e di chiamata, e gli indirizzi dei dati che si incontrano nelle istruzioni di load e store. Ciascuno di tali indirizzi relativi passa attraverso due passi di manipolazione da parte del processore. In primo luogo, il valore contenuto nel registro base è sommato all'indirizzo relativo per produrre un indirizzo assoluto; successivamente, l'indirizzo risultante è confrontato con il valore del registro limite. Se l'indirizzo è dentro ai limiti, allora

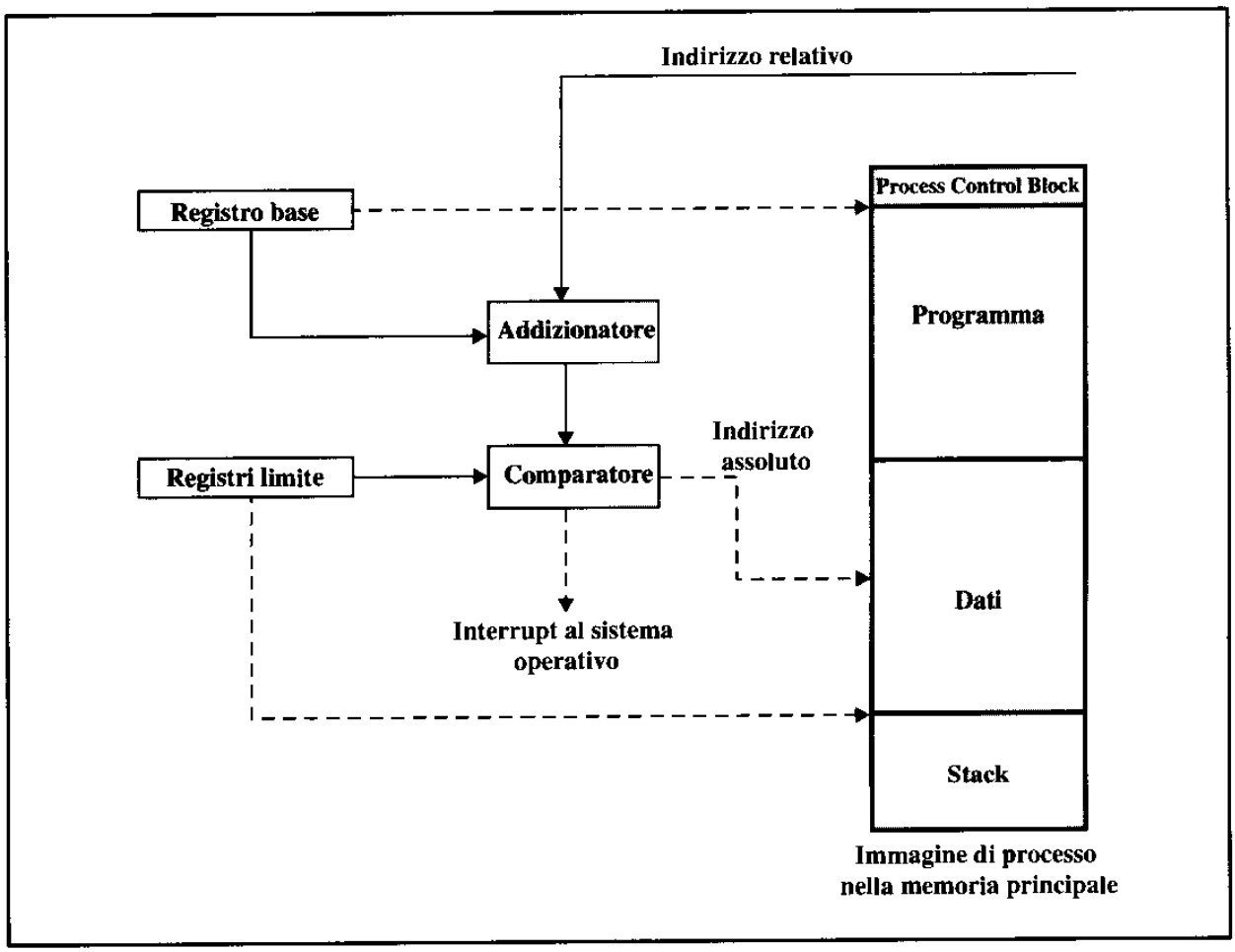


Figura 7.8 *Supporto hardware per la rilocazione*

l'esecuzione dell'istruzione può procedere; altrimenti, è generato un interrupt al sistema operativo, che deve rispondere all'errore in qualche maniera.

Lo schema di Figura 7.8 permette ai programmi di essere trasferiti in memoria e sul disco durante l'esecuzione; inoltre, costituisce un meccanismo di protezione: ogni immagine di processo è isolata tramite il contenuto dei registri base e limite, e protetta da accessi indesiderati degli altri processi.

7.3 Paginazione

Sia il partizionamento di dimensione fissa, sia quello di dimensione variabile usano la memoria in modo inefficiente; il primo provoca la frammentazione interna, il secondo la frammentazione esterna. Si supponga, invece, che la memoria principale sia partizionata in blocchi relativamente piccoli di uguale dimensione fissa, e che ogni processo sia a sua volta diviso in piccoli "pacchetti" della stessa dimensione. Il pacchetto di un processo, detto **pagina** (*page*), può essere assegnato al blocco di memoria disponibile, detto **frame** o page frame. Mostreremo in questa sezione che lo spazio di memoria sprecato per ogni processo è dovuto alla

frammentazione interna, limitata ad una porzione dell'ultima pagina del processo: non c'è quindi frammentazione esterna.

La Figura 7.9 illustra l'uso di pagine e frame. Ad un certo punto, alcuni frame in memoria sono usati mentre altri sono liberi; il sistema operativo mantiene una lista di frame liberi. Il processo A, memorizzato su disco, comprende quattro pagine. Quando viene il momento di

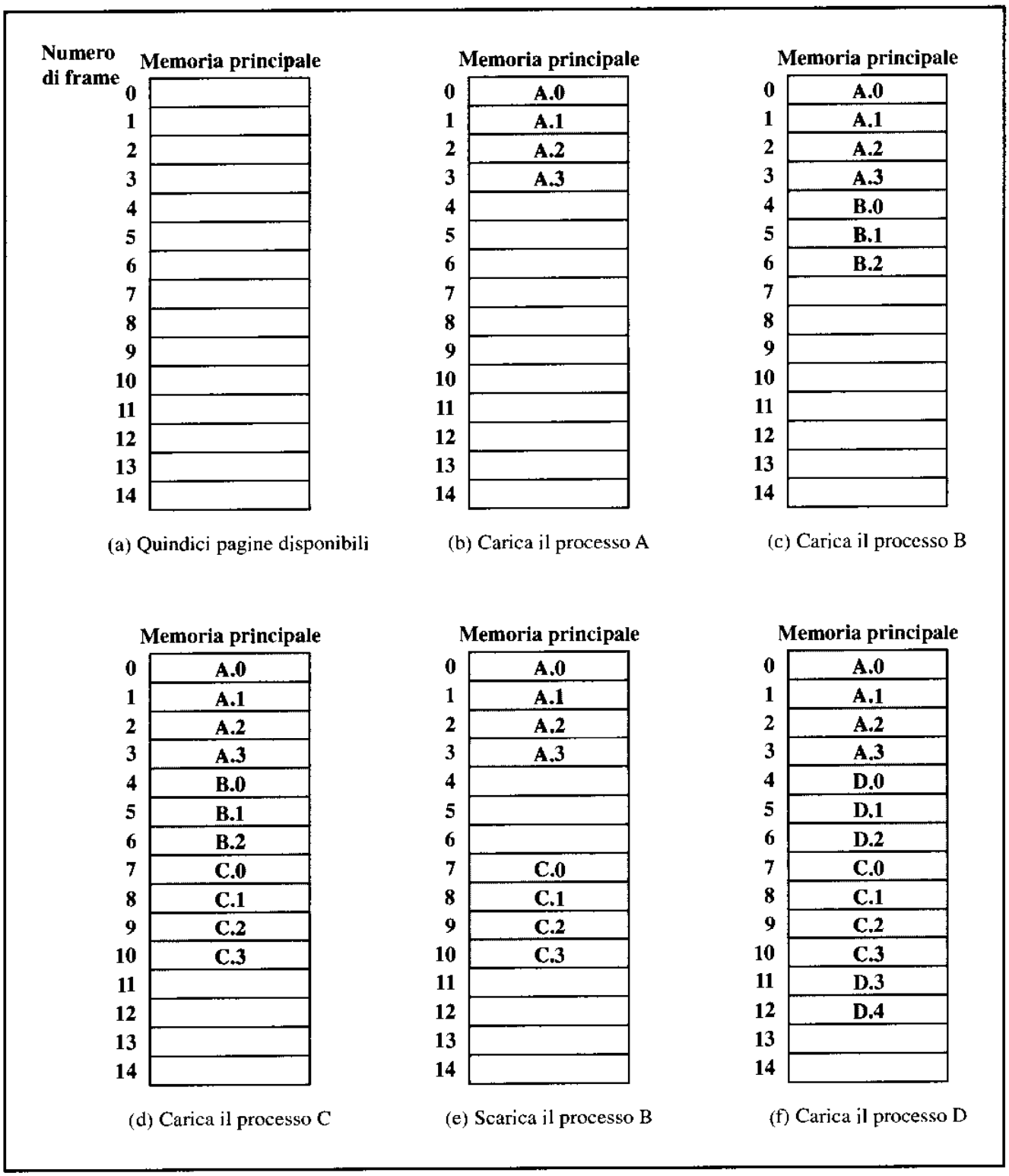


Figura 7.9 *Assegnazione di pagine dei processi ai frame liberi*

caricare questo processo, il sistema operativo trova quattro frame liberi e carica le quattro pagine del processo A nei quattro frame (Figura 7.9b). Il processo B, che comprende tre pagine, e il processo C, che comprende quattro pagine, sono caricati successivamente. Poi il processo B è sospeso ed è trasferito fuori dalla memoria principale; più tardi, tutti i processi nella memoria principale sono bloccati, e il sistema operativo deve caricare un nuovo processo, il processo D, che comprende cinque pagine.

Ora si supponga, come in questo esempio, che non ci siano abbastanza frame contigui liberi per contenere il processo. Questo impedisce al sistema operativo di caricare D? La risposta è no, perché noi possiamo ancora una volta usare il concetto di indirizzo logico, anche se un solo registro base non sarà più sufficiente: il sistema operativo mantiene una **page table** (*tavola delle pagine*) per ogni processo. La page table contiene la locazione del frame corrispondente ad ogni pagina del processo; all'interno del programma, ogni indirizzo logico si compone di un numero di pagina e di un offset dentro la pagina. Ricordiamo che, nel caso di una partizione semplice, un indirizzo logico rappresenta la locazione di una parola relativamente all'inizio del programma; il processore lo traduce in un indirizzo fisico. Con la paginazione, la traduzione dell'indirizzo da logico a fisico è ancora fatta dall'hardware del processore, che accede alla page table del processo corrente. In presenza di un indirizzo logico (numero della pagina, offset), il processore usa la page table per produrre un indirizzo fisico (numero del frame, offset).

Continuando il nostro esempio, le cinque pagine del processo D sono caricate nei frame 4, 5, 6, 11 e 12. La Figura 7.10 mostra le varie page table in questo punto: una page table contiene un'entry per ogni pagina del processo, così la tavola si può facilmente indicizzare con il numero della pagina (a partire da pagina 0). Ogni entry della page table contiene il numero del frame nella memoria principale, che contiene la pagina corrispondente, se è presente. Inoltre, il sistema operativo mantiene un'unica "lista dei frame liberi", contenente tutti i frame di memoria principale che al momento non sono occupati e sono disponibili per caricarvi pagine.

Perciò la paginazione semplice, come è qui descritta, è simile al partizionamento fisso; la differenza è che, con la paginazione, le partizioni sono abbastanza piccole, e che un programma può occupare più partizioni, non necessariamente contigue.

Per rendere questo schema di paginazione conveniente, imponiamo che la dimensione della pagina, e quindi la dimensione del frame, sia una potenza di 2. In questo caso l'indirizzo relativo, che è definito relativamente all'origine del programma, e l'indirizzo logico, espresso da un

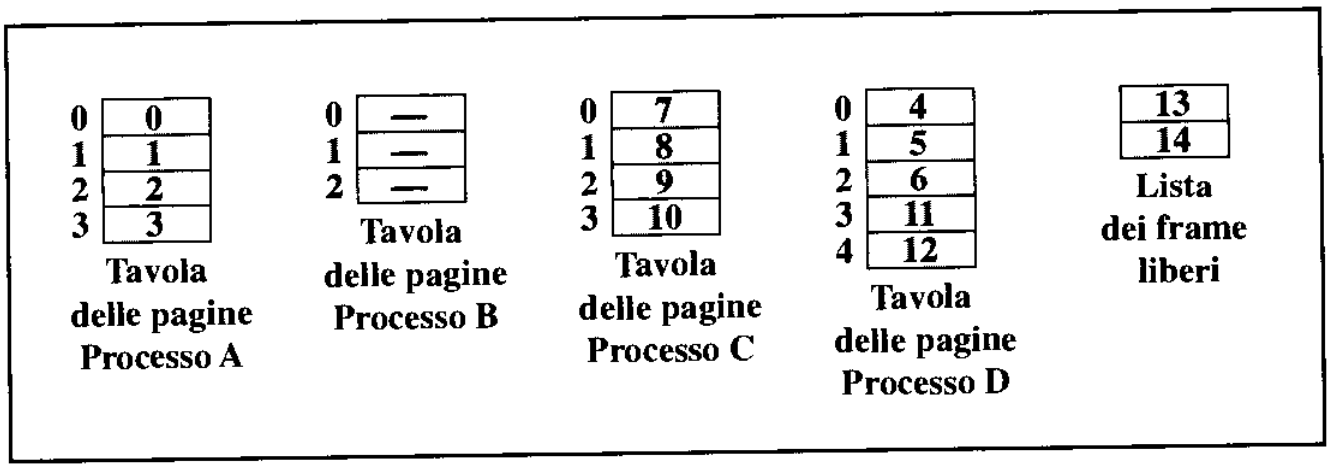


Figura 7.10 *Strutture dati per l'esempio di Figura 7.9 all'istante (f)*

numero di pagina e un offset, sono gli stessi. Un esempio è mostrato in Figura 7.11: in questo esempio, sono usati indirizzi a 16 bit e la dimensione della pagina è di 1K = 1024 byte. L'indirizzo relativo 1502, in forma binaria, è 0000010111011110; con una pagina di dimensione 1KB, è necessario un campo di offset di 10 bit, lasciando 6 bit per il numero della pagina. Perciò un programma può comporsi di un massimo di 2^6 = 64 pagine, da un kilobyte ciascuna. Come mostra la Figura 7.11b, l'indirizzo relativo 1502 corrisponde ad un offset di 478 (0111011110) su pagina 1 (000001), cioè lo stesso numero a 16 bit, 0000010111011110.

Le conseguenze dell'uso di una dimensione della pagina che è una potenza di 2 sono duplici. Per prima cosa, lo schema di indirizzamento logico è trasparente al programmatore, all'assemblatore e al linker. Ogni indirizzo logico (numero di pagina, offset) di un programma è identico al suo indirizzo relativo. Secondariamente, è relativamente semplice eseguire per hardware dinamicamente, a tempo di esecuzione, la traduzione degli indirizzi. Si consideri un indirizzo di $n + m$ bit, dove gli n bit più a sinistra sono il numero della pagina e gli m bit più a destra sono l'offset. Nel nostro esempio (Figura 7.11b), $n = 6$ e $m = 10$. I passi seguenti effettuano la traduzione dell'indirizzo:

- Si estrae il numero della pagina, cioè gli n bit più a sinistra dell'indirizzo logico.

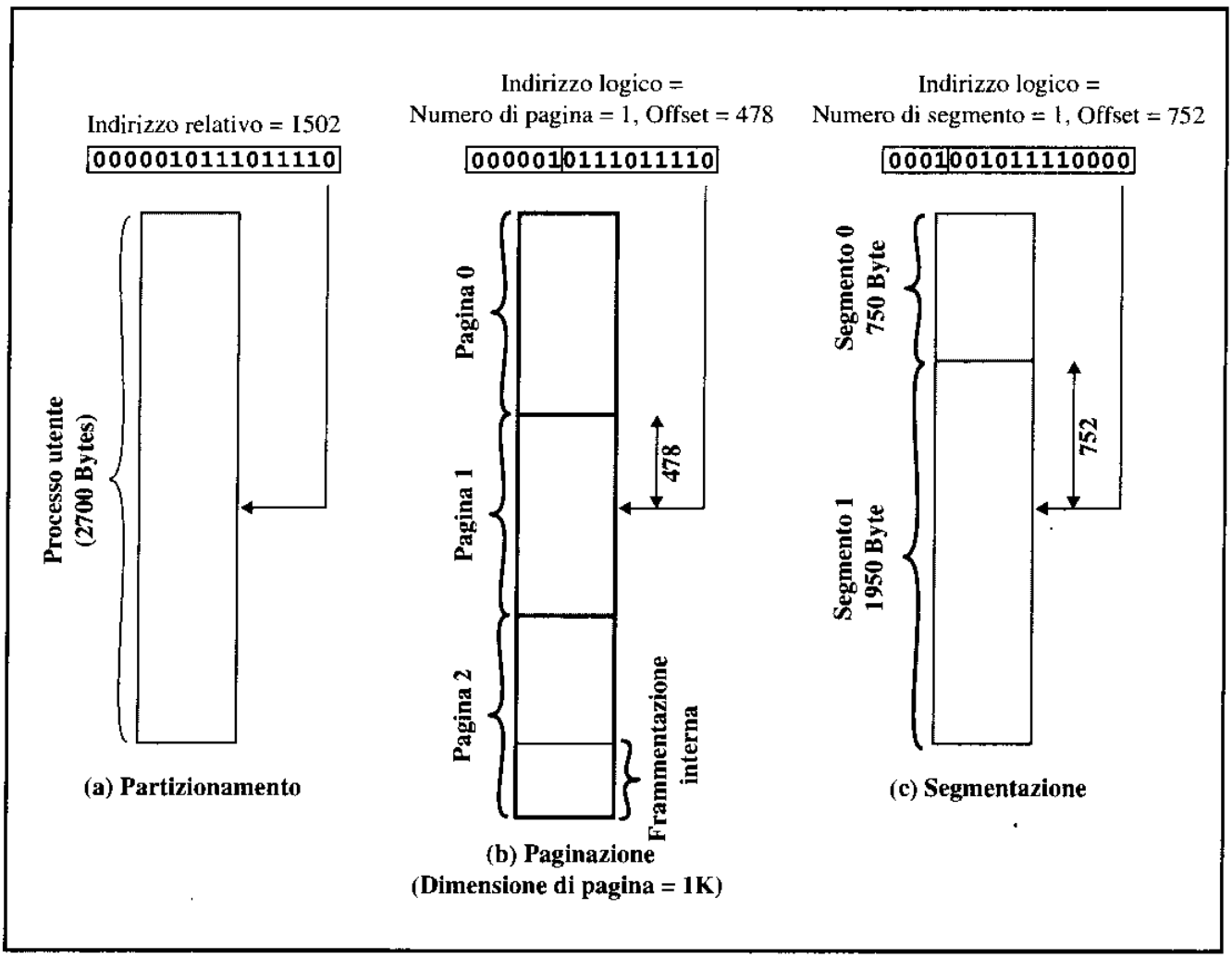


Figura 7.11 *Indirizzi logici*

- Si usa il numero della pagina come un indice nella page table del processo per trovare il numero del frame, k.
- L'indirizzo fisico iniziale del frame è $k \times 2^m$, e l'indirizzo fisico del byte indirizzato è tale numero più l'offset. Questo indirizzo fisico non deve essere calcolato; è costruito semplicemente concatenando i bit del numero di frame e dell'offset.

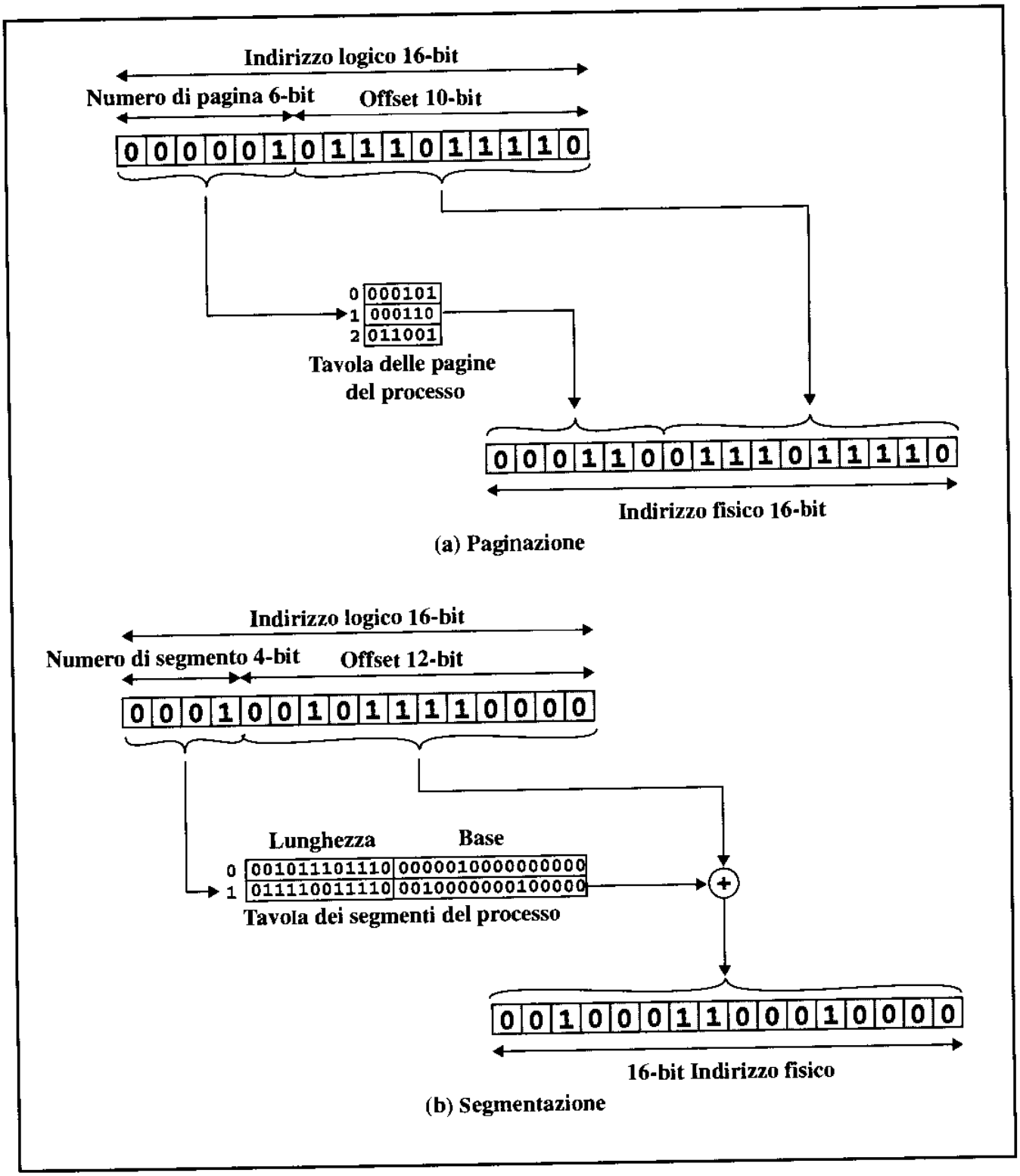


Figura 7.12 *Esempi di traduzione da indirizzo logico ad indirizzo fisico*

Nel nostro esempio, abbiamo l'indirizzo logico 0000010111011110, che è la pagina numero 1 e offset 478. Si supponga che questa pagina risieda nel frame 6 della memoria principale (in notazione binaria 000110): allora, l'indirizzo fisico è il frame numero 6, offset 478 = 0001100111011110 (Figura 7.12a).

Riassumendo, con la paginazione semplice, la memoria principale è divisa in molti frame piccoli di uguale dimensione, ed ogni processo è diviso in pagine aventi la dimensione del frame; processi più piccoli richiedono meno pagine, processi più grandi ne richiedono di più. Quando un processo è caricato in memoria, tutte le sue pagine sono caricate nei frame liberi, ed è creata la sua page table; tale tecnica risolve molti problemi che il partizionamento non risolveva.

7.4 Segmentazione

In alternativa alla paginazione, un programma utente può anche essere suddiviso tramite la segmentazione. In questo caso, il programma e i dati ad esso associati sono divisi in un certo numero di **segmenti**. Non è richiesto che tutti i segmenti di tutti i programmi siano della stessa lunghezza, però esiste un limite massimo di lunghezza del segmento. Come avviene per la paginazione, un indirizzo logico che usa la segmentazione è composto da due parti: il numero del segmento e un offset.

Poiché i segmenti usati possono avere lunghezza diversa, la segmentazione è simile al partizionamento dinamico. In assenza di uno schema di overlay o dell'uso di memoria virtuale, bisogna caricare tutti i segmenti di un programma in memoria per eseguirlo. La differenza che intercorre con il partizionamento dinamico è che, con la segmentazione, un programma può utilizzare più di una partizione, e le partizioni usate possono essere non contigue. La segmentazione elimina quindi la frammentazione interna ma, come il partizionamento dinamico, presenta il problema della frammentazione esterna; tuttavia, poiché un processo è suddiviso in pezzi più piccoli, la frammentazione esterna è minore.

Mentre la paginazione è invisibile al programmatore, la segmentazione è solitamente visibile, ed è conveniente per organizzare programmi e dati. Generalmente, il programmatore o il compilatore assegneranno il programma e i dati a segmenti diversi; per programmare modularmente, il programma o i dati possono essere ulteriormente frammentati in molti segmenti. Lo svantaggio di questo meccanismo è che il programmatore deve prestare attenzione alla dimensione massima del segmento.

Un'altra conseguenza dell'uso di segmenti di dimensione differente è che non esiste nessuna relazione semplice tra l'indirizzo logico e l'indirizzo fisico. Analogamente a quanto avviene per la paginazione, uno schema semplice per la segmentazione usa una tavola dei segmenti per ogni processo e una lista dei blocchi liberi nella memoria principale. Ogni entry nella tavola dei segmenti indicherà l'indirizzo iniziale del segmento corrispondente nella memoria principale; fornirà anche la lunghezza del segmento, per assicurarsi che non siano usati indirizzi non validi. Quando un processo diventa Running, l'indirizzo della sua tavola dei segmenti è caricato in un registro speciale usato dall'hardware del gestore della memoria. Si consideri un indirizzo di $n + m$ bit, dove gli n bit più a sinistra sono il numero del segmento e gli m bit più a destra sono

l'offset; nel nostro esempio (Figura 7.11c), $n = 4$ e $m = 12$. Perciò la dimensione massima del segmento è $2^{12} = 4096$. Per la traduzione dell'indirizzo, sono necessari i passi seguenti:

- Si estrae il numero del segmento dagli n bit più a sinistra dell'indirizzo logico.
- Si usa il numero del segmento come un indice nella segment table del processo per trovare l'indirizzo fisico iniziale del segmento.
- Si confronta l'offset, rappresentato dagli m bit più a destra, con la lunghezza del segmento; se l'offset è maggiore della lunghezza, l'indirizzo non è valido.
- L'indirizzo fisico desiderato è la somma dell'indirizzo fisico iniziale del segmento con l'offset.

Nel nostro esempio, abbiamo l'indirizzo logico 0001001011110000, che è il segmento numero 1 con offset 752. Si supponga che questo segmento risieda nella memoria principale a partire dall'indirizzo fisico 0010000000100000. Allora l'indirizzo fisico è 0010000000100000 + 001011110000 = 0010001100010000 (Figura 7.12b).

Riassumendo, con la segmentazione semplice, un processo è diviso in un certo numero di segmenti non necessariamente di uguale dimensione; quando un processo è portato in memoria, tutti i suoi segmenti sono caricati nelle regioni disponibili in memoria, ed è creata una segment table.

7.5 Sommario

Uno dei compiti più importanti e complessi di un sistema operativo è la gestione della memoria. La gestione della memoria implica il trattamento della memoria come una risorsa da allocare e condividere tra un certo numero di processi attivi. Per usare efficientemente le funzionalità del processore e dell'I/O, si consiglia di mantenere quanti più processi possibile nella memoria principale. Inoltre, nello sviluppo dei programmi, si cerca di liberare i programmatori dalle limitazioni sulle dimensioni.

Gli strumenti fondamentali utilizzati per la gestione della memoria sono la paginazione e la segmentazione; con la paginazione, ogni processo è diviso in pagine relativamente piccole di dimensione fissata, mentre la segmentazione divide i programmi in pacchetti di dimensione variabile. È anche possibile combinare la segmentazione e la paginazione in un unico schema per la gestione della memoria.

7.6 Letture raccomandate

I libri sui sistemi operativi consigliati nella Sezione 2.9 sono esaurienti per la parte riguardante la gestione di memoria. Poiché il partizionamento è stato sostituito dalle tecniche di memoria virtuale, la maggior parte del libri accenna semplicemente alla questione. Due dei più

completi e interessanti sono [MILE92] e [HORN89], mentre si può trovare un'approfondita trattazione sulle strategie di partizionamento in [KNUT97].

In molti libri sulle tecniche di sviluppo dei programmi, sull'architettura del computer e sui sistemi operativi, si possono trovare trattazioni in merito al link ed al caricamento; se si cerca invece una trattazione più completa si può consultare [BECK90].

L'appendice 7A è stata creata seguendo la linea di [SCHN85], che fornisce una chiara introduzione di base. [PINK89] è un ottimo riassunto, che schematizza i passi che precedono il link ed il caricamento nella creazione di un modulo oggetto, mentre il problema del link dinamico trova ampia descrizione in [BIC88] e [KRAK88] con particolare riferimento all'approccio di Multics.

BECK90 Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1990.

BIC88 Bic, L., e Shaw, A. *The logical Design of Operating Systems*, Second Edition. Englewood Cliffs, NJ: Prentice Hall, 1988.

HORN89 Horner, D. *Operating Systems: Concepts and Applications*. Glenview, IL: Scott, Foresman, &Co., 1989.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition*. Reading, MA: Addison-Wesley, 1997.

KRAK88 Krakoviak, S., e Beeson, D. *Principles of Operating Systems*. Cambridge, MA: MIT Press, 1988.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New York:McGraw-Hill, 1992.

PINK89 Pinkert, J., e Wear, L. *Operating Systems: Concepts, Policies, and Mechanism*. Englewood Cliffs, NJ: Prentice Hall, 1989.

SCHN85 Schneider, G. *The Principles of Computer Organization*. New York: Wiley 1985.

7.7 Problemi

7.1 Nella Sezione 2.3 sono state elencate le cinque responsabilità del gestore di memoria, e nella Sezione 7.1 sono stati elencati i cinque requisiti. Provare che ogni elenco comprende tutte la voci presenti nell'altra.

7.2 Si consideri uno schema di partizionamento dinamico. Provare che, in media, la memoria contiene un numero di blocchi liberi pari a metà di quello dei segmenti.

7.3 Per implementare i vari algoritmi di allocazione, discussi per il partizionamento dinamico, nella Sezione 7.2, deve essere mantenuta una lista dei blocchi di memoria liberi. Per ognuno dei tre metodi discussi (best-fit, first-fit, next-fit), qual è il tempo medio di ricerca?

7.4 Un altro algoritmo di allocazione è il worst-fit. In questo caso, è usato il blocco più grande per inserire in memoria un processo. Discutere i pro e i contro di questo metodo, confrontandolo con i tre metodi presentati. Qual è il tempo medio di ricerca per il worst-fit?

7.5 Usando il buddy system, si allochi un blocco da un megabyte.

a. Riportare i risultati della seguente sequenza di richieste in una tabella simile alla tabella di Figura 7.6: Richiesti 70; Richiesti 35; Richiesti 80; Restituito A; Richiesti 60; Restituito B; Restituito D; Restituito C.

b. Tracciare la rappresentazione ad albero binario dopo Restituito B.

7.6 Si consideri un buddy system nel quale un particolare blocco, nell'allocazione corrente, ha indirizzo 011011110000.

a. Se il blocco è di dimensione 4, qual è l'indirizzo binario del suo compagno?

b. Se il blocco è di dimensione 16, qual è l'indirizzo del suo compagno?

7.7 Sia $buddy_k(x)$ l'indirizzo del compagno del blocco di dimensione 2^k il cui indirizzo è x. Scrivere l'espressione generale di $buddy_k(x)$.

7.8 La sequenza di Fibonacci è così definita:

$$F_0 = 0, \qquad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n, \qquad n \geq 0$$

a. Si può usare questa sequenza per definire un buddy system?

b. Quale sarebbe il vantaggio di questo sistema, se confrontato con il buddy system binario descritto in questo capitolo?

7.9 Durante l'esecuzione di un programma, il processore incrementa il contenuto del registro di istruzioni (program counter) di una parola dopo ogni istruzione, ma modifica il contenuto di quel registro se incontra un trasferimento di controllo o una chiamata di istruzione che causa il proseguimento dell'esecuzione in un altro punto del programma. Si consideri quindi la Figura 7.8: esistono due alternative per indirizzare l'istruzione:

- Tenere un indirizzo relativo nel registro di istruzioni e tradurre dinamicamente l'indirizzo usando il registro di istruzioni come input; quando si esegue una istruzione di trasferimento di controllo o una chiamata, l'indirizzo relativo generato da tali istruzioni è caricato nel registro di istruzioni.
- Tenere un indirizzo assoluto nel registro di istruzione; quando si esegue una istruzione di trasferimento di controllo o una chiamata, l'indirizzo è tradotto dinamicamente, ed i risultati sono memorizzati nel registro di istruzione.

Qual è l'approccio migliore?

7.10 Si consideri una memoria in cui i segmenti contigui $S1$, $S2$, ..., Sn sono messi nell'ordine di creazione da una estremità all'altra, come suggerito dalla seguente figura:

S1	S2	•••	S*n*	Buco

Quando il segmento $Sn+1$ è creato, esso è allocato immediatamente dopo Sn anche se sono già stati eliminati alcuni segmenti precedenti. Quando il limite tra i segmenti (in uso o cancellati) e il blocco libero raggiunge l'estremo sinistro della memoria, i segmenti in uso sono compattati.

a. Provare che la percentuale di tempo F impiegata per compattare obbedisce alla seguente diseguaglianza:

$$F \geq \frac{1-f}{1+kf} \quad \text{dove} \quad k = \frac{t}{2s} - 1$$

definendo:

s = lunghezza media del segmento, in parole

t = durata media di un segmento, in termini di numero di riferimenti di memoria

f = percentuale di memoria inutilizzata in condizioni di equilibrio

Suggerimento: trovare la velocità media a cui il limite attraversa la memoria, supponendo che la copiatura di una singola parola richiede almeno due riferimenti di memoria.

b. Calcolare F per $f = 0.2$, $t = 1000$ e $s = 50$.

Appendice 7A Caricamento e link

Il primo passo nella creazione di un processo attivo è caricare un programma nella memoria principale e creare un'immagine di processo (Figura 7.13). La Figura 7.14 mostra uno scenario tipico per la maggior parte dei sistemi: l'applicazione si compone di un certo numero di moduli compilati o assemblati, sotto forma di moduli oggetto. Su di essi, l'azione di *collegamento* o *link* risolve ogni riferimento tra i moduli; inoltre, risolve anche i riferimenti alle funzioni di libreria, che possono essere incorporate nei programmi, o gestite come codice condiviso fornito dal sistema operativo a tempo di esecuzione. In questa appendice, accenneremo alle caratteristiche principali dei programmi per il link e il caricamento. Per chiarezza nella presentazione, si comincia con la descrizione di un'operazione di caricamento in cui è coinvolto un solo modulo di programma e non è richiesto alcun link.

Caricamento

In Figura 7.14, il programma di caricamento mette il modulo di caricamento nella memoria principale, a partire dalla locazione x; durante tale operazione, si soddisfa il requisito di indirizzamento illustrato nella Figura 7.1. In generale, possono essere considerati tre approcci:

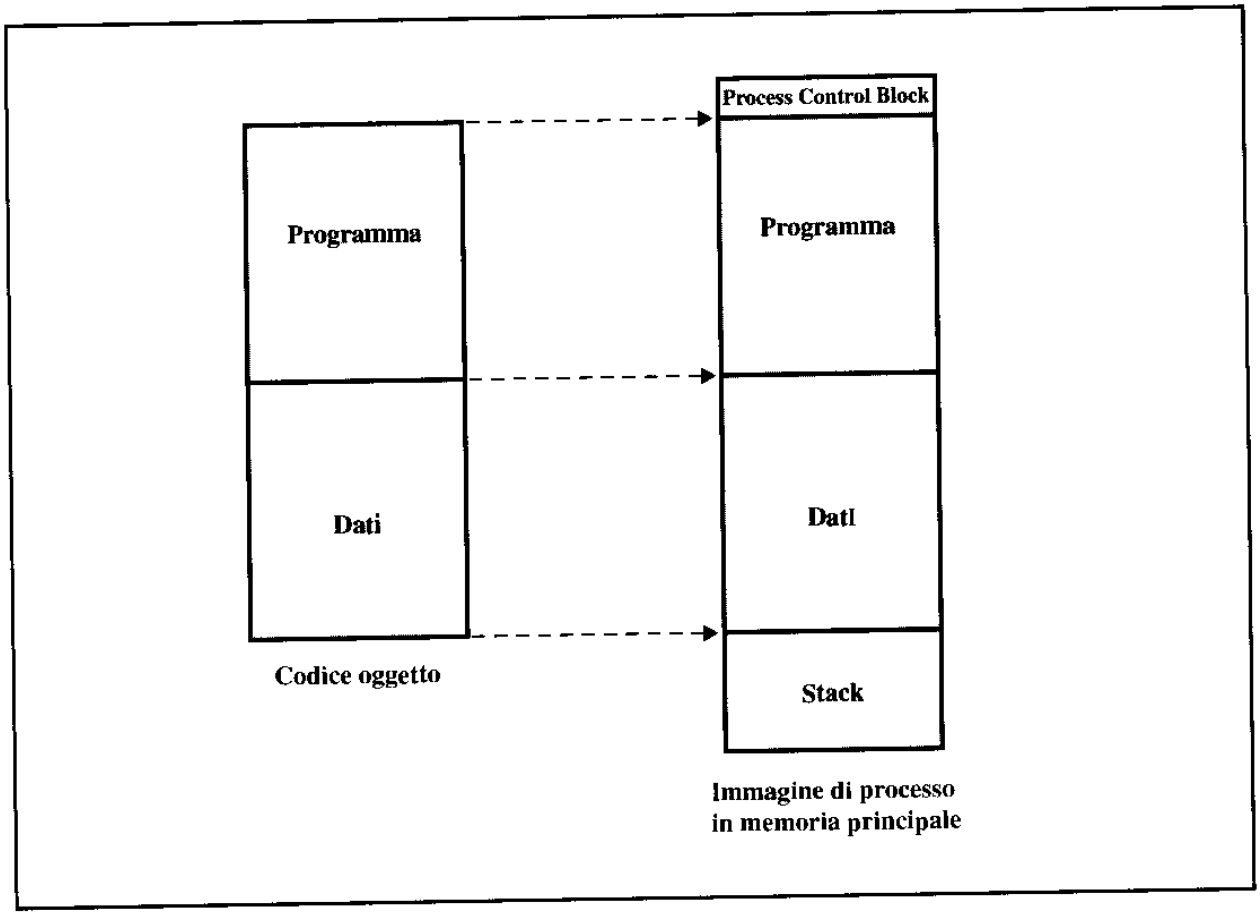


Figura 7.13 *La funzione di caricamento*

- Caricamento assoluto
- Caricamento rilocabile
- Caricamento dinamico a tempo di esecuzione.

Caricamento assoluto

Un programma di caricamento assoluto richiede che un dato modulo di caricamento sia sempre caricato nella stessa locazione di memoria principale. Perciò, nel modulo di caricamento, tutti i riferimenti devono essere fatti ad indirizzi specifici, o assoluti, nella memoria principale. Per esempio, se x nella Figura 7.14 è la locazione 1024, allora la prima parola nel modulo di caricamento, destinata a quella regione di memoria, ha indirizzo 1024.

L'assegnazione di specifici valori di indirizzo a riferimenti di memoria all'interno di un programma può essere fatta dal programmatore sia al momento della compilazione sia dell'assemblaggio (Tabella 7.2a). Vi sono diversi svantaggi nel primo approccio: innanzi tutto, ogni programmatore dovrebbe conoscere la strategia di allocazione dei moduli all'interno della memoria principale; in secondo luogo, se sono fatte modifiche al programma, con inserimenti o cancellazioni nel corpo del modulo, allora tutti gli indirizzi vanno alterati. Quindi, è preferibile consentire di esprimere simbolicamente i riferimenti a memoria all'interno dei programmi, e poi

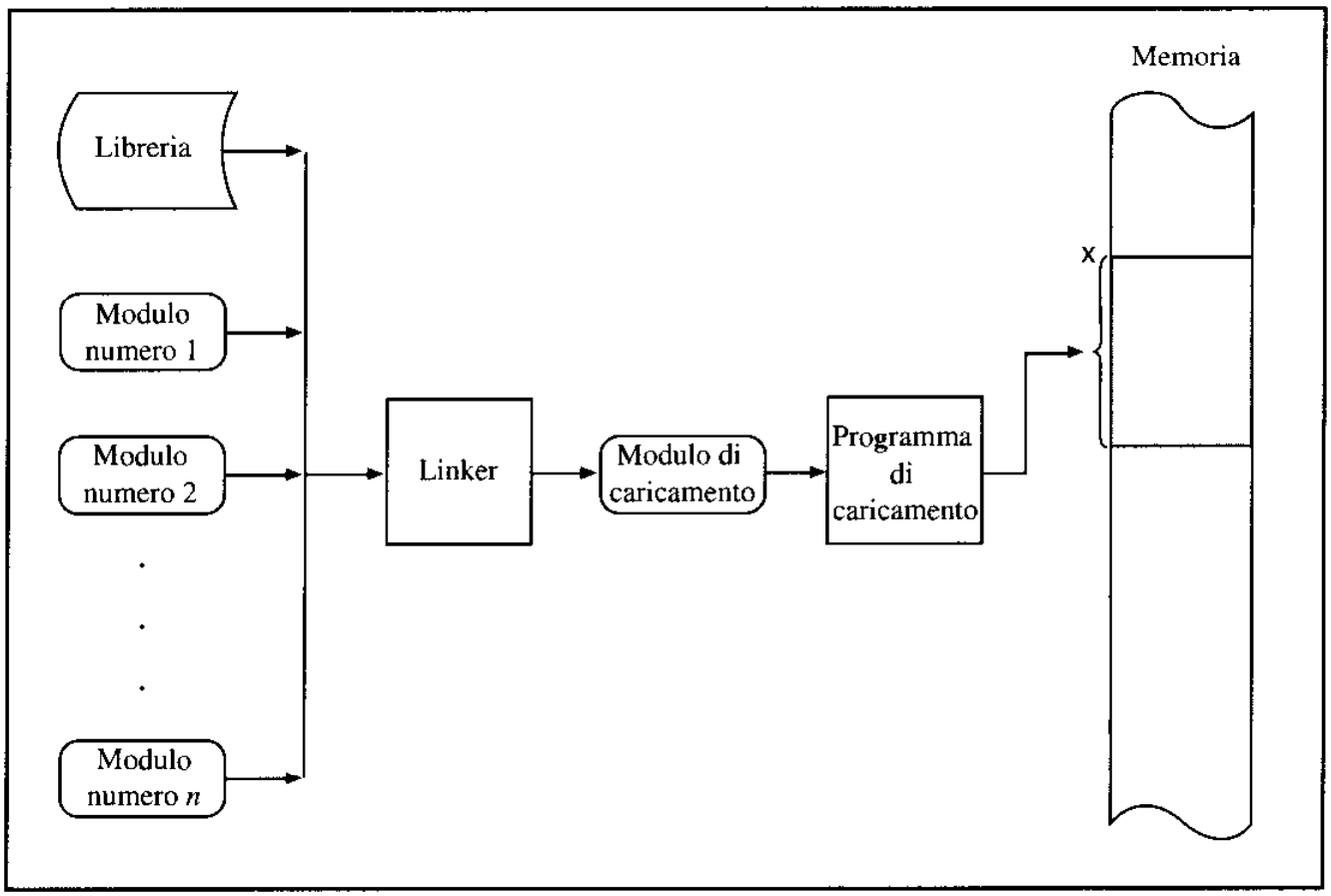


Figura 7.14 *Uno scenario di caricamento*

risolvere quei riferimenti simbolici a tempo di compilazione o assemblaggio, come illustrato nella Figura 7.15. Ogni riferimento a un'istruzione o dato è inizialmente rappresentato da un simbolo; durante la traduzione del modulo, per darlo in input ad un programma di caricamento assoluto, l'assemblatore o il compilatore convertiranno tutti questi riferimenti in indirizzi specifici (in questo esempio, per un modulo che deve essere caricato a partire dalla locazione 1024).

Caricamento rilocabile

Lo svantaggio di assegnare i riferimenti di memoria, legandoli ad un indirizzo specifico prima del caricamento, è che il modulo di caricamento risultante può essere allocato in una sola regione della memoria principale. Quando molti programmi condividono la memoria principale, non è consigliabile decidere in anticipo in quale regione della memoria deve essere caricato un particolare modulo; è meglio prendere tale decisione al momento del caricamento, perciò è meglio avere un modulo di caricamento che possa essere allocato in qualunque parte della memoria.

Per soddisfare questo nuovo requisito, l'assemblatore o il compilatore non genera indirizzi assoluti in memoria principale non attuali, bensì indirizzi relativi a qualche punto conosciuto, ad esempio l'inizio del programma. Questa tecnica è illustrata in Figura 7.15b. L'inizio del modulo di caricamento è assegnato all'indirizzo relativo 0, e tutti gli altri riferimenti di memoria all'interno del modulo sono espressi relativamente all'inizio del modulo.

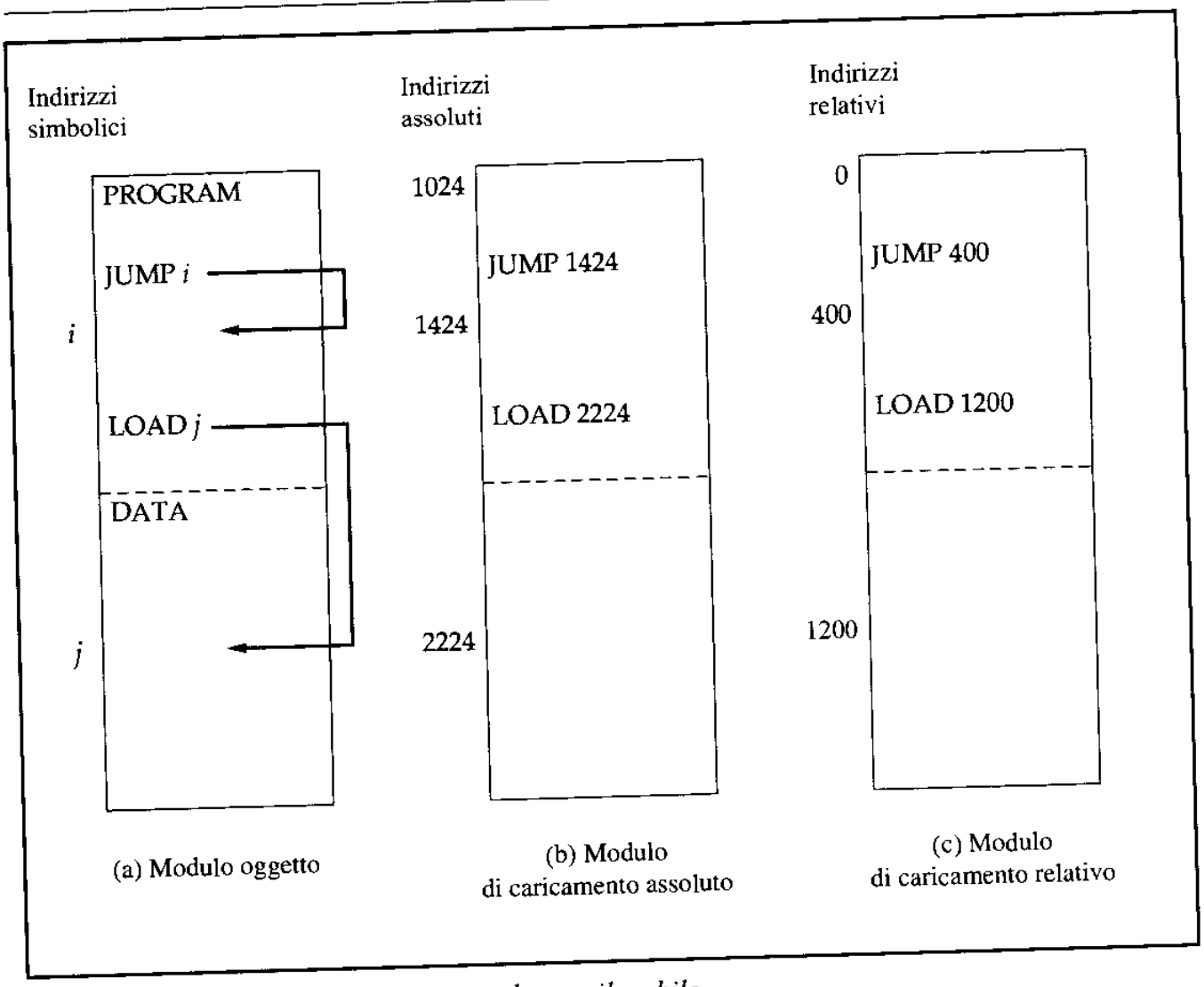


Figura 7.15 *Moduli di caricamento assoluto e rilocabile*

Con tutti i riferimenti di memoria espressi in formato relativo, diventa semplice per il programma di caricamento allocare il modulo nella locazione desiderata. Se il modulo deve essere caricato a partire dalla locazione x, allora il programma di caricamento deve semplicemente aggiungere x ad ogni riferimento di memoria, quando lo carica in memoria. Perché questo sia possibile, il modulo di caricamento deve contenere informazioni che dicano al programma di caricamento dove sono i riferimenti e come devono essere interpretati (solitamente saranno relativi all'inizio del programma, ma potrebbero anche essere relativi a qualche altro punto nel programma, come la locazione corrente). Queste informazioni sono preparate dal compilatore o dall'assemblatore e sono solitamente dette dizionario di rilocazione.

Caricamento dinamico a tempo di esecuzione

I programmi di caricamento rilocabile sono diffusi e forniscono ovvi benefici rispetto ai programmi di caricamento assoluto; tuttavia, in un ambiente in multiprogrammazione, anche senza memoria virtuale, lo schema di caricamento rilocabile è inadeguato. Abbiamo già citato la necessità di trasferire le immagini dei processi dentro e fuori dalla memoria, per massimizzare

Tabella 7.2 *Collegamento degli indirizzi*

(a) Programma di caricamento

Tempo di collegamento	Funzione
Tempo di programmazione	Tutti gli indirizzi fisici effettivi sono direttamente specificati dal programmatore nello stesso programma
Tempo di compilazione o assemblaggio	Il programma contiene riferimenti a indirizzi simbolici, che sono trasformati in indirizzi fisici effettivi dal compilatore o dall'assemblatore
Tempo di caricamento	Il compilatore o l'assemblatore produce indirizzi relativi. Il programma di caricamento traduce questi indirizzi in indirizzi assoluti al momento del caricamento del programma
Tempo di esecuzione	Il programma caricato contiene indirizzi relativi, che l'hardware del processore traduce dinamicamente in indirizzi assoluti.

(b) Linker

Tempo di link	Funzione
Tempo di programmazione	Non sono permessi riferimenti a programmi esterni o a dati. Il programmatore deve mettere nel programma il codice sorgente per tutti i sottoprogrammi usati.
Tempo di compilazione o assemblaggio	L'assemblatore deve recuperare il codice sorgente di ogni subroutine, a cui fa riferimento il processo, e assemblarlo unitariamente.
Creazione del modulo di caricamento	Tutti i moduli oggetto sono stati assemblati usando gli indirizzi relativi. Questi moduli sono collegati e tutti i riferimenti sono relativi all'origine del modulo di caricamento finale.
Tempo di caricamento	Non si risolvono i riferimenti esterni fino a che il modulo non va caricato nella memoria principale. A quel punto, i moduli riferiti tramite link dinamici sono agganciati al modnlo di caricamento, e l'intero pacchetto è caricato nella memoria principale o nella memoria virtuale
Tempo di esecuzione	I riferimenti esterni non sono risolti fino a che il processore non esegue la chiamata esterna. A quel punto, s'interrompe il processo, e si collega il modulo di caricamento con il programma che ha richiesto

l'utilizzazione del processore; per massimizzare l'utilizzazione della memoria principale, bisognerebbe trasferire le immagini dei processi in locazioni differenti, in momenti diversi. Perciò, un programma, una volta caricato, può essere messo su disco e poi reinserito in una locazione diversa, e questo sarebbe impossibile se i riferimenti di memoria fossero stati legati ad indirizzi assoluti al tempo del caricamento iniziale.

L'alternativa è di rimandare il calcolo di un indirizzo assoluto fino a che esso non è necessario a tempo di esecuzione; a questo scopo, il modulo di caricamento è caricato nella memoria principale con tutti i riferimenti di memoria in formato relativo (Figura 7.15c): non si calcola l'indirizzo assoluto fino a che non è eseguita un'istruzione che lo utilizza. Per essere sicuri che questa funzione non peggiori le prestazioni, la deve eseguire un hardware specializzato del processore, piuttosto che il software. Questo hardware è stato descritto nella Sezione 7.2.

Il calcolo dell'indirizzo dinamico fornisce la massima flessibilità: un programma può essere caricato in qualunque regione della memoria principale; in seguito, la sua esecuzione può essere interrotta, ed il programma può essere tolto dalla memoria principale, per essere poi ricaricato ad una diversa locazione.

Link

La funzione di link prende in input una collezione di moduli oggetto, per produrre un modulo di caricamento, che si compone di un insieme integrato di programma e moduli di dati, pronti per essere passati al programma di caricamento. In ogni modulo oggetto, possono esserci riferimenti a locazioni in altri moduli; ciascuno di tali riferimenti deve essere espresso simbolicamente nel modulo oggetto prima del link. La funzione di link crea un unico modulo di caricamento, allocando ad indirizzi contigui tutti i moduli oggetto. Ogni riferimento simbolico tra i moduli deve essere sostituito con un riferimento ad una locazione all'interno del modulo di caricamento unico, che li contiene tutti. Per esempio, il modulo A nella Figura 7.16a contiene una chiamata ad una procedura del modulo B. Quando questi moduli sono riuniti nel modulo di caricamento, questo riferimento simbolico al modulo B è sostituito da un riferimento alla locazione specifica, a partire dal punto di ingresso in B nel modulo di caricamento.

Linkage Editor

La natura del link degli indirizzi dipenderà dal tipo di modulo di caricamento che deve essere creato e da quando avviene il link (Tabella 7.2b). Se, come accade solitamente, si desidera un modulo di caricamento rilocabile, allora il link è solitamente fatto nel seguente modo: ogni modulo oggetto compilato o assemblato è creato con riferimenti relativi all'inizio del modulo oggetto; tutti questi moduli sono messi insieme in un unico modulo di caricamento rilocabile, cioè contenente tutti i riferimenti relativi all'origine del modulo di caricamento. Questo modulo può essere usato come input per il caricamento rilocabile, o per il caricamento dinamico a tempo di esecuzione.

Un programma che produce un modulo di caricamento rilocabile è spesso chiamato **linkage editor** (editore dei link). La Figura 7.16 illustra la funzione del linkage editor.

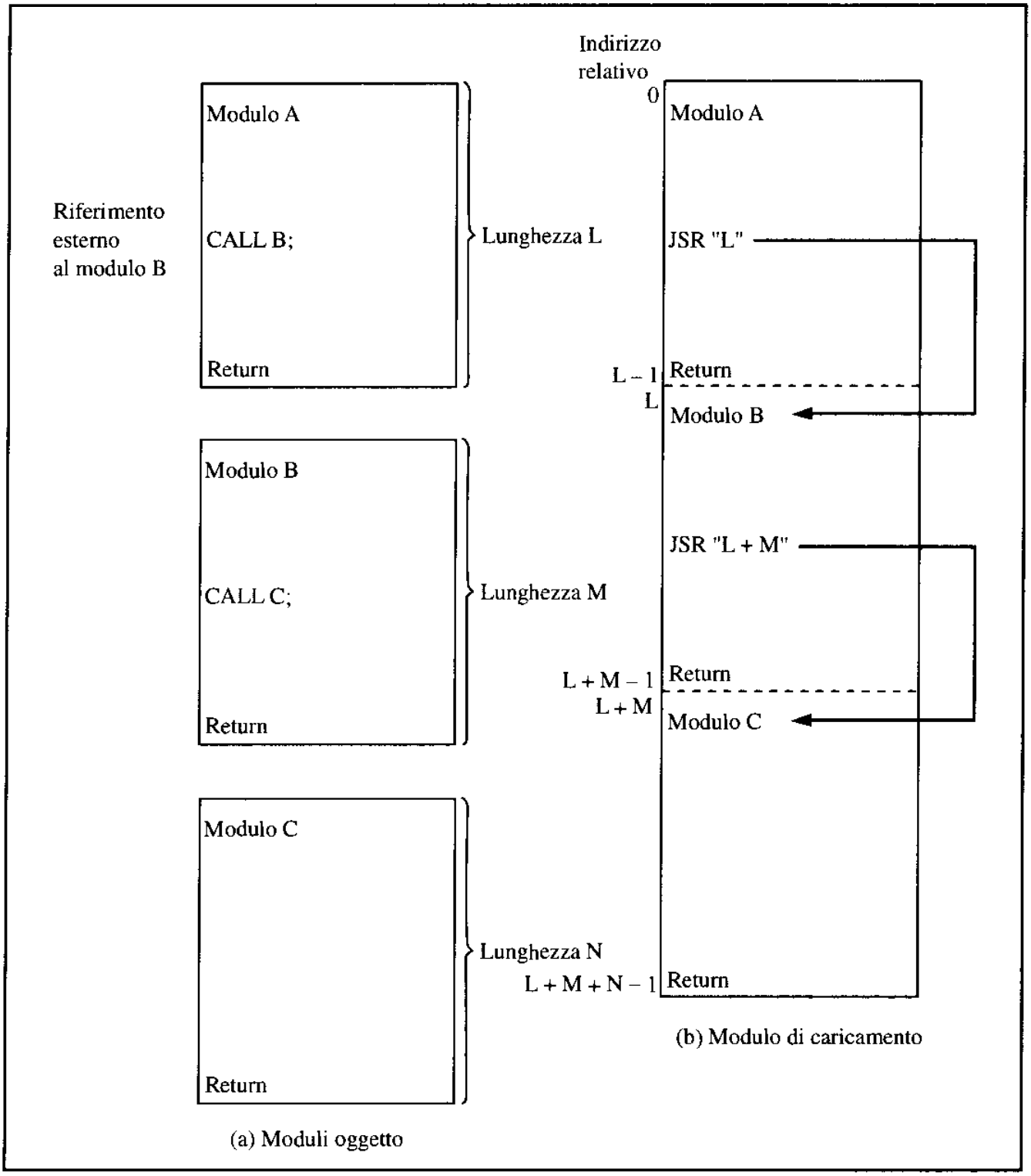


Figura 7.16 *La funzione di link*

Link dinamico

Come per il caricamento, è possibile rinviare alcune funzioni di link. Il termine *dynamic linkage* (link dinamico) si riferisce a una tecnica in cui si rinvia il link di alcuni moduli esterni, anche dopo che è stato creato il modulo di caricamento. Perciò il modulo di caricamento contie-

ne riferimenti non risolti ad altri programmi, che saranno risolti al momento del caricamento oppure a tempo di esecuzione.

Per il link dinamico al momento del caricamento è necessario seguire i seguenti passi: il modulo di caricamento (l'applicazione), che deve essere caricato, è letto in memoria, ed ogni riferimento a un modulo esterno (modulo target) costringe il programma di caricamento a trovare tale modulo target, caricarlo e modificare il riferimento con un indirizzo relativo all'inizio del modulo applicativo. Questo approccio ha diversi vantaggi rispetto al cosiddetto caricamento statico:

- Diventa più facile incorporare versioni modificate o aggiornate del modulo target, che può essere un'utility del sistema operativo o qualche altra routine di uso generale. Con il link statico, un cambiamento di tale modulo di supporto potrebbe richiedere un nuovo link su tutta l'applicazione. Questo non solo è inefficiente, ma può anche essere impossibile, in alcune circostanze: per esempio, nel campo dei personal computer, la maggior parte del software commerciale è realizzato in forma di modulo di caricamento; le versioni sorgente e oggetto non sono distribuite.
- Avere il modulo target in un file di link dinamico prepara la strada alla condivisione automatica del codice. Il sistema operativo può accorgersi che più di un'applicazione sta usando lo stesso codice target perché lo ha caricato e ne ha fatto il link; può quindi usare tale informazione per caricarne una sola copia, accessibile ad entrambe le applicazioni, piuttosto che doverne caricare una copia per ogni applicazione.

Diventa più facile per i progettisti di software estendere indipendentemente le funzionalità di un sistema operativo largamente usato, come OS/2. Un progettista può presentare una nuova funzione, che può essere utile ad una varietà di applicazioni, e renderla disponibile come modulo di link dinamico.

Con il **link dinamico a tempo di esecuzione**, una parte dei riferimenti è risolta a tempo di esecuzione, infatti, i riferimenti esterni a moduli target rimangono nel modulo di caricamento. Quando è chiamata una funzione del modulo assente, il sistema operativo alloca il modulo, lo carica e fa il link al modulo chiamante.

Abbiamo visto come il caricamento dinamico permetta ad un intero modulo di caricamento di essere spostato; comunque la struttura del modulo è statica, rimanendo inalterata durante l'esecuzione del processo e da un'esecuzione alla successiva. In alcuni casi, non è possibile determinare prima dell'esecuzione quale modulo oggetto sarà richiesto. Questa situazione è tipica delle applicazioni che elaborano transazioni, come i sistemi di prenotazione per compagnie aeree o le applicazioni bancarie: la natura della transazione determina quali moduli del programma siano necessari, e questi sono caricati e agganciati al programma principale. Il vantaggio dell'uso di un tale link dinamico è che non è necessario allocare la memoria per moduli di programma che non sono utilizzati; tale capacità è usata congiuntamente ai sistemi di segmentazione.

È possibile un ulteriore raffinamento: una applicazione non necessita di sapere i nomi di tutti i moduli o punti di ingresso che possono essere chiamati. Ad esempio, un programma di grafica può essere scritto per lavorare con una varietà di plotter, ognuno dei quali è gestito da un diverso

driver. L'applicazione può conoscere il nome del plotter installato attualmente nel sistema da un altro processo, o cercandolo nel file di configurazione: ciò consente all'utente dell'applicazione di installare un nuovo plotter, che non esisteva nel momento in cui è stata scritta l'applicazione.

CAPITOLO 8

LA MEMORIA VIRTUALE

Nel Capitolo 7 sono stati introdotti i concetti di paginazione e segmentazione e sono stati analizzati i loro difetti. In questo capitolo sarà affrontato il problema della memoria virtuale. L'analisi di quest'argomento è difficile poiché la gestione della memoria ha una relazione complessa con l'hardware del processore e il software del sistema operativo. Quindi, ci concentreremo prima sugli aspetti hardware della memoria virtuale, osservando l'uso della paginazione, della segmentazione e della combinazione delle due, per poi affrontare i problemi coinvolti nella progettazione della memoria virtuale nei sistemi operativi.

8.1 Strutture hardware e di controllo

Confrontando la segmentazione e la paginazione semplici con il partizionamento fisso e dinamico, sono evidenti le basi per una svolta fondamentale nella gestione della memoria. Le caratteristiche chiave di paginazione e segmentazione sono rappresentate da:

1. Tutti i riferimenti di memoria all'interno di un processo sono indirizzi logici, che sono tradotti dinamicamente in indirizzi fisici a tempo d'esecuzione. Questo significa che un processo può essere caricato in memoria principale e scaricato in modo da occupare diverse regioni di memoria in diversi momenti durante l'esecuzione
2. Un processo può essere suddiviso in un certo numero di pezzi (pagine o segmenti), che non devono necessariamente essere contigui in memoria al momento dell'esecuzione. Quest'opzione è permessa dalla combinazione della traduzione dinamica di indirizzi a tempo d'esecuzione, con l'uso di una tabella dei segmenti o delle pagine.

La chiave di volta è rappresentata dalla seguente osservazione: se le due caratteristiche citate sono presenti, allora non è necessario che tutte le pagine o tutti i segmenti di un processo siano nella memoria principale durante l'esecuzione. Se il pezzo (segmento o pagina) che contiene l'istruzione successiva e il pezzo che contiene la successiva locazione dei dati sono nella memoria principale, allora almeno per un certo periodo l'esecuzione può procedere.

Vediamo quindi come realizzare tutto ciò. Per ora si parlerà in termini generali, e si userà il termine pezzo per riferirsi sia alla pagina sia al segmento, a seconda di quale delle due tecniche è usata. Si supponga di dover caricare un nuovo processo in memoria, il sistema operativo comincia a caricare solo uno o alcuni pezzi, per includere il pezzo contenente l'inizio del programma. La porzione di un processo che è al momento nella memoria principale è chiamata **resident set** (insieme residente) del processo. Non appena il processo comincia la sua esecuzione, le cose proseguono tranquillamente fintanto che i riferimenti a memoria sono in locazioni presenti nel resident set. Usando la tabella dei segmenti o delle pagine, il processore è sempre in grado di determinare se questo è vero. Se il processore incontra un indirizzo logico che non è nella memoria principale, genera un'interruzione, che indica un fallimento dell'accesso in memoria. Il sistema operativo mette il processo interrotto in stato Blocked e prende il controllo. Affinché questo processo possa procedere in seguito, il sistema operativo avrà bisogno di riportare nella memoria principale il pezzo del processo che contiene l'indirizzo logico che ha causato il fallimento di accesso: a questo scopo, il sistema operativo emette una richiesta di lettura da disco. Dopo che la richiesta di I/O è stata emessa, il sistema operativo può mandare in esecuzione un altro processo mentre è eseguito l'I/O di disco. Una volta che il pezzo desiderato è stato caricato in memoria, è generato un interrupt di I/O, restituendo il controllo al sistema operativo, che riporta a Ready il processo precedentemente bloccato.

Sorge spontaneo chiedersi quale sia l'efficienza di questa manovra, nella quale un processo può essere mandato in esecuzione, ma può essere interrotto per il semplice fallimento del caricamento di tutti i pezzi necessari al processo. Per adesso, si rimandi la discussione di questo quesito e si accetti l'assicurazione che si può avere efficienza. Invece, sarebbe meglio focalizzare le implicazioni della nuova strategia: ce ne sono due, la seconda più sorprendente della prima, ed entrambe portano ad un migliore uso del sistema:

1. Più processi possono essere mantenuti nella memoria principale. Poiché noi stiamo caricando solo alcuni pezzi di un processo, c'è spazio per più processi. Questo conduce ad un uso più efficiente del processore, perché è più probabile che almeno un processo, in un gruppo numeroso, sia in stato Ready in ogni momento.

2. È possibile che un processo sia più grande della memoria principale, eliminando così una delle più fondamentali restrizioni della programmazione. Senza lo schema che è stato discusso, un programmatore deve stare molto attento alla quantità di memoria disponibile. Se il programma che sta scrivendo è troppo grande, il programmatore deve escogitare dei modi per strutturarlo in pezzi, da caricare separatamente con una strategia di overlay. Con la memoria virtuale basata sulla paginazione o sulla segmentazione, quel lavoro è lasciato al sistema operativo e all'hardware. Il programmatore ha quindi a che fare con un'immensa memoria, di dimensione pari allo spazio disco; il sistema operativo carica automaticamente i pezzi di un processo nella memoria principale come richiesto.

Poiché un processo è eseguito solo nella memoria principale, quella memoria è chiamata **memoria reale**. Ma un programmatore o un utente percepisce una memoria potenzialmente più grande, quella allocata sul disco, detta **memoria virtuale.** La memoria virtuale permette un'efficiente multiprogrammazione e libera l'utente dagli inutili vincoli della memoria principale. La Tabella 8.1 riassume le caratteristiche della paginazione e della segmentazione, con e senza l'uso della memoria virtuale.

Località e memoria virtuale

I benefici della memoria virtuale sono accattivanti, ma si ha a che fare con uno schema pratico? Un tempo vi fu un acceso dibattito su questo punto, ma l'esperienza di numerosi sistemi operativi ha dimostrato, oltre ogni dubbio, che la memoria virtuale funziona. Di conseguenza, è diventata una componente essenziale della maggior parte dei sistemi operativi contemporanei.

Per capire perché la memoria virtuale sia stata oggetto di tanto dibattere, si esamini ancora una volta il compito del sistema operativo per realizzare la memoria virtuale. Si consideri un grosso processo, che si compone di un lungo programma più numerose array di dati. Per un qualunque breve periodo, l'esecuzione può essere limitata ad una piccola sezione del programma (ad esempio una subroutine), e l'accesso a uno o forse due array di dati. Se è così, allora sarebbe chiaramente dispendioso caricare dozzine di pezzi di quel processo, se in realtà solo pochi pezzi saranno utilizzati prima che il processo sia sospeso e copiato su disco: è possibile fare un uso migliore della memoria caricando solo i pochi pezzi necessari. Allora se il programma salta ad un'istruzione o si riferisce a un dato non presente nella memoria principale, si genera un errore, per comunicare al sistema operativo che è necessario caricare il pezzo desiderato.

Perciò, in ogni momento, solo pochi pezzi di un dato processo sono presenti in memoria. e perciò più processi possono essere mantenuti in memoria. Inoltre, si risparmia tempo perché i pezzi inutilizzati non sono trasferiti né dentro né fuori della memoria; comunque, il sistema deve gestire in modo intelligente questo schema. In una condizione di stabilità, praticamente tutta la memoria principale è occupata dai pezzi dei processi, in modo che il processore e il sistema operativo abbiano accesso diretto a più processi possibili, e quando il sistema operativo carica un pezzo di processo nella memoria principale, deve prima toglierne un altro. Se ne toglierà uno che sta per essere usato, dovrà recuperarlo quasi immediatamente: se si verificheranno troppi eventi di questo tipo, si giungerà ad una condizione detta **thrashing:** il processore passa la maggior parte del tempo trasferendo pezzi piuttosto che eseguendo istruzioni. L'eliminazione del thrashing è stata oggetto di molte ricerche negli anni '70, che hanno portato ad una varietà di algoritmi, complessi ma efficaci. In sostanza, il sistema operativo cerca di indovinare, in base ai passi precedenti, quali pezzi saranno probabilmente meno usati nel seguito.

Questo ragionamento è basato sul principio di località, che è stato introdotto nel capitolo 1 (in particolare nell'Appendice 1A). In poche parole, il principio di località afferma che i riferimenti al programma o ai dati all'interno di un processo tendono a raggrupparsi; quindi, l'ipotesi che solo pochi pezzi di un processo saranno necessari in un breve periodo è valida. Inoltre, sarebbe possibile formulare ipotesi intelligenti su quali pezzi di un processo saranno necessari nel prossimo futuro, il che eliminerebbe il thrashing.

Un modo per confermare il principio di località è osservare le prestazioni dei processi in un ambiente di memoria virtuale. La Figura 8.1 è un diagramma abbastanza famoso che illustra

Tabella 8.1 *Caratteristiche di Paginazione e Segmentazione*

Paginazione pura	Segmentazione pura	Paginazione della memoria virtuale	Memoria virtuale con segmentazione
La memoria principale è partizionata in un insieme di piccoli blocchi di dimensione fissa chiamati frame	La memoria principale non è partizionata	La memoria principale è partizionata in un insieme di piccoli blocchi di dimensione fissa chiamati frame	La memoria principale non è partizionata
Il programma è spezzato in pagine dal compilatore o dal gestore della memoria	I segmenti del programma sono specificati dal programmatore o dal compilatore (cioè, la decisione è presa dal programmatore)	Il programma è spezzato in pagine dal compilatore o dal gestore della memoria	I segmenti del programma sono specificati dal programmatore al compilatore (cioè la decisione è presa dal programmatore)
Frammentazione interna ai frame	No frammentazione interna	Frammentazione interna ai frame	Non c'è frammentazione interna
No frammentazione esterna	Frammentazione esterna	No frammentazione esterna	Frammentazione esterna
Il sistema operativo fa uso di una tabella delle pagine per ogni processo che mostra in che frame si trova la pagina	Il sistema operativo fa uso di una tabella dei segmenti per ogni processo che mostra l'indirizzo iniziale e la lunghezza d'ogni segmento.	Il sistema operativo fa uso di una tabella delle pagine per ogni processo che mostra in che frame si trova la pagina	Il sistema operativo deve mantenere una tabella dei segmenti per ogni processo, che indichi l'indirizzo di caricamento e la lunghezza di ogni segmento
Il sistema operativo fa uso di una lista dei frame liberi.	Il sistema operativo fa uso di una lista di spazi liberi nella memoria principale	Il sistema operativo fa uso di una lista dei frame liberi	Il sistema operativo deve mantenere una lista dei buchi liberi nella memoria principale
Il processore usa il numero di pagina e l'offset per calcolare l'indirizzo assoluto.	Il processore usa il numero di segmento e l'offset per calcolare l'indirizzo assoluto.	Il processore usa il numero di pagina e l'offset per calcolare l'indirizzo assoluto	Il processore usa il numero di segmento e l'offset per calcolare l'indirizzo assoluto
Tutte le pagine di un processo devono essere nella memoria principale affinché il processo possa essere eseguito, a meno che non siano usati overlay	Tutti i segmenti di un processo devono essere nella memoria principale affinché il processo possa essere eseguito, a meno che non siano usati overlay	Non tutte le pagine di un processo devono necessariamente essere nei frame della memoria principale affinché il processo possa essere eseguito. Le pagine possono essere lette quando sono necessarie	Non tutti i segmenti di un processo devono essere in frame della memoria principale perché il processo possa andare in esecuzione. I segmenti possono essere caricati quando necessari
		La lettura di una pagina nella memoria principale può richiedere la scrittura su disco di un'altra pagina	Leggere un segmento in memoria principale può richiedere la scrittura di uno o più segmenti sul disco

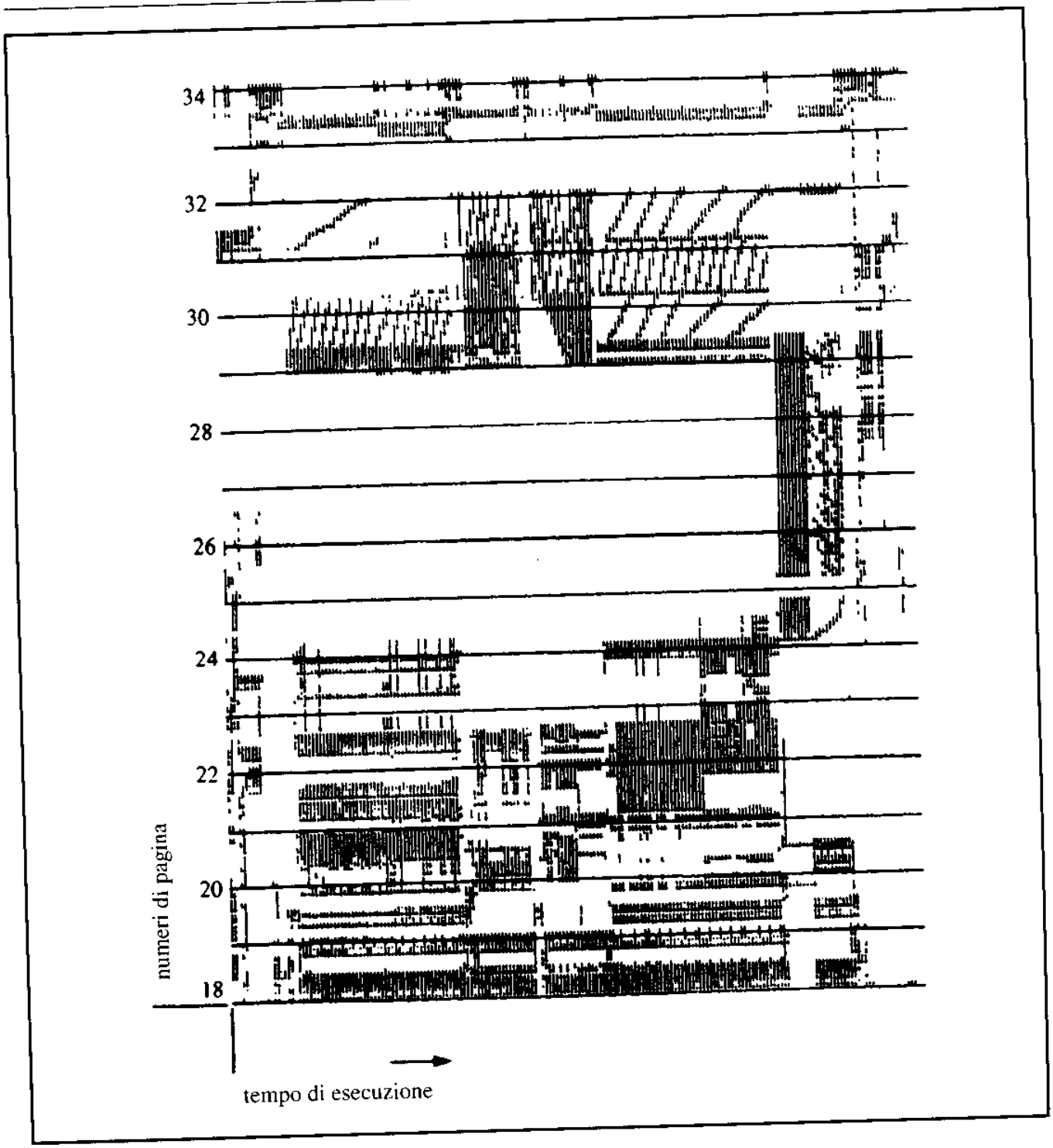


Figura 8.1 *Comportamento della paginazione*

clamorosamente il principio di località [HATF72]; si noti che, per la durata del processo, i riferimenti sono limitati ad un sottoinsieme di pagine.

Perciò il principio di località suggerisce che la memoria virtuale può funzionare. Affinché la memoria virtuale sia pratica ed efficiente, sono necessari due ingredienti. Innanzi tutto, deve esserci un supporto hardware per utilizzare gli schemi di paginazione e/o segmentazione. Secondariamente, il sistema operativo deve includere del software per gestire gli spostamenti delle pagine o dei segmenti tra la memoria secondaria e la memoria principale. In questa sezione, si

esamineranno gli aspetti hardware e le necessarie strutture di controllo, che sono create e mantenute dal sistema operativo, ma sono usate dall'hardware per la gestione della memoria. La prossima sezione prenderà in esame i problemi del sistema operativo.

Paginazione

Il termine *memoria virtuale* è solitamente associato a sistemi che utilizzano la paginazione, sebbene sia usata anche la memoria virtuale basata sulla segmentazione, di cui si parlerà in seguito. L'uso della paginazione per ottenere la memoria virtuale fu documentato per la prima volta sul computer Atlas [KILB62] e presto divenne di diffuso uso commerciale.

Nella discussione sulla paginazione semplice, abbiamo indicato che ogni processo ha la sua tabella delle pagine (*page table*) e, quando tutte le sue pagine sono caricate nella memoria principale, la tabelle delle pagine per un processo è creata e caricata nella memoria principale. Ogni entry della tabella delle pagine contiene il numero del frame della pagina corrispondente nella memoria principale. Lo stesso strumento, una tabella delle pagine, è necessario quando noi consideriamo uno schema di memoria virtuale basato sulla paginazione: anche in questo caso, si associa un'unica tabella delle pagine ad ogni processo. In questo caso, invece, le entry della tabella delle pagine diventano più complicate (Figura 8.2a): poiché solo alcune delle pagine di un processo possono stare nella memoria principale, è necessario un bit in ogni entry della tabella delle pagine per indicare se la pagina corrispondente è presente (P) o meno nella memoria principale. Se il bit indica che la pagina è in memoria, allora l'entry contiene anche il numero di frame di quella pagina.

Un altro bit di controllo è necessario nell'entry della tabella delle pagine: il bit di modifica (M) che indica se i contenuti della pagina corrispondente sono stati alterati da quando la pagina è stata caricata nella memoria principale. Se non ci sono stati cambiamenti, non è necessario riscrivere su disco la pagina quando viene il momento di toglierla dal frame che occupa. Possono essere presenti altri bit di controllo: per esempio, se la protezione o la condivisione è gestita a livello di pagina saranno richiesti bit per quello scopo.

Struttura della tabella delle pagine

Il meccanismo basilare per la lettura di una parola dalla memoria riguarda la traduzione di un indirizzo virtuale, o logico (che si compone di un numero di pagina e un offset), in un indirizzo fisico (che si compone di un numero di frame e un offset), usando una tabella delle pagine. Poiché la tabella delle pagine è di lunghezza variabile, secondo la dimensione del processo, non si può pensare di tenerla in registri; viceversa, deve essere nella memoria principale per accedervi. La Figura 8.3 suggerisce un'implementazione hardware: quando un particolare processo è in esecuzione, un registro conserva l'indirizzo di partenza della tabella delle pagine di quel processo. Il numero di pagina di un indirizzo virtuale è usato come indice in quella tabella per cercare il corrispondente numero di frame, combinarlo con l'offset dell'indirizzo virtuale e produrre l'indirizzo reale desiderato.

Nella maggior parte dei sistemi, è presente una tabella delle pagine per processo; ma ogni processo può occupare enormi quantità di memoria virtuale. Per esempio, nell'architettura VAX, ogni processo può avere fino a $2^{31} = 2$ gigabyte di memoria virtuale; usando pagine di $2^9 = 512$

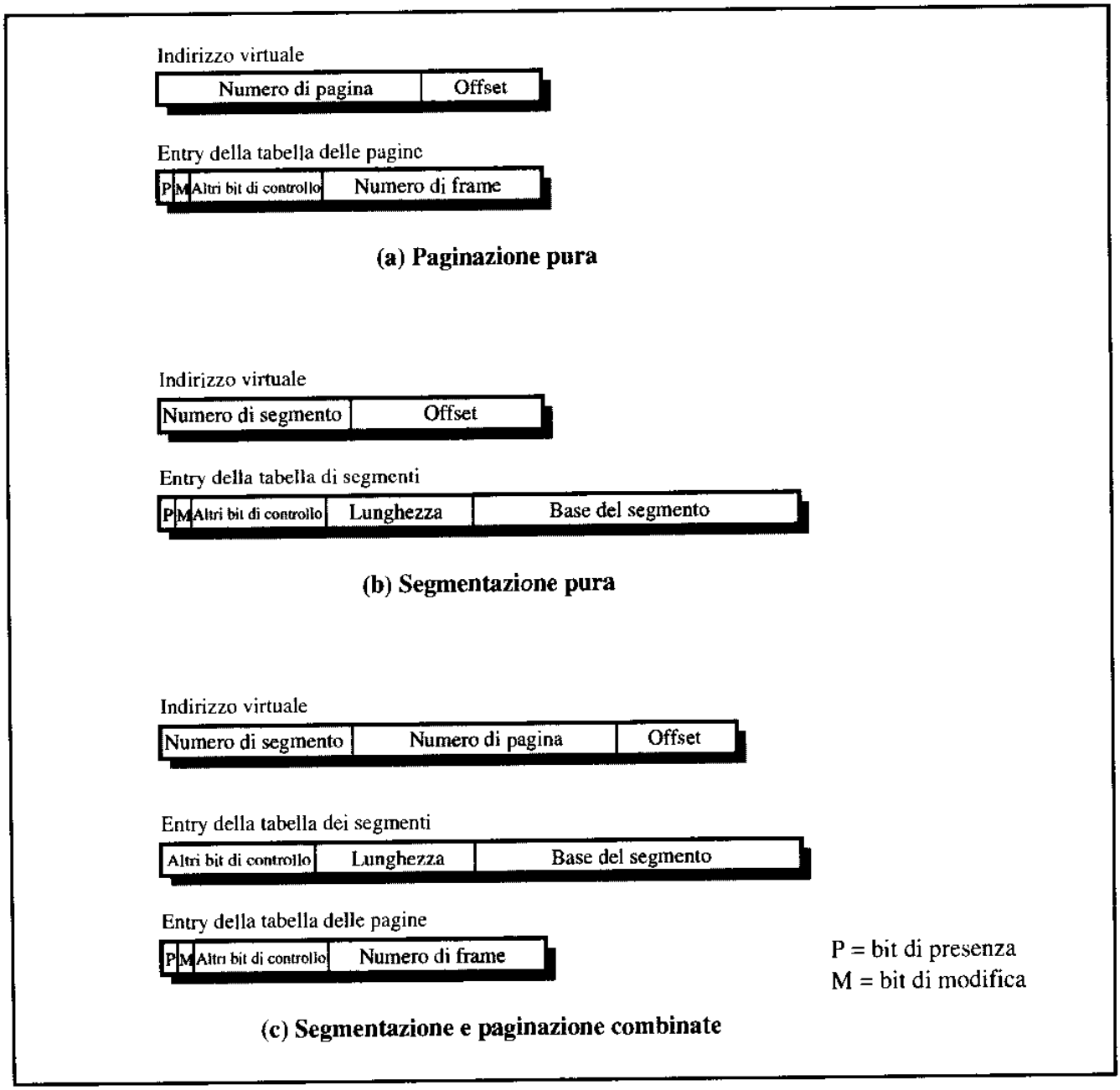


Figura 8.2 *Formati tipici per la gestione della memoria*

byte, ci vogliono 2^{22} entry di tabella delle pagine per ogni processo: in tal caso, la quantità di memoria dedicata solamente alle tabelle delle pagine potrebbe essere troppo alta. Per superare questo problema, molti schemi di memoria virtuale memorizzano le tabelle delle pagine nella memoria virtuale piuttosto che nella memoria reale. Ciò significa che le tabelle delle pagine sono soggette alla paginazione, come le altre pagine, e che quando un processo è in esecuzione, almeno una parte della sua tabella delle pagine deve essere nella memoria principale: l'entry della pagina correntemente in esecuzione deve essere nella memoria principale. Alcuni processori fanno uso di uno schema a due livelli per organizzare grandi tabelle delle pagine: si usa infatti una directory di pagine, in cui ogni entry punta ad una tabella delle pagine. In questo modo, se la lunghezza della directory delle pagine è X, e se la massima lunghezza di una tabella delle pagine è Y, allora un processo può essere composto di X x Y pagine. Tipicamente, la massima lunghez-

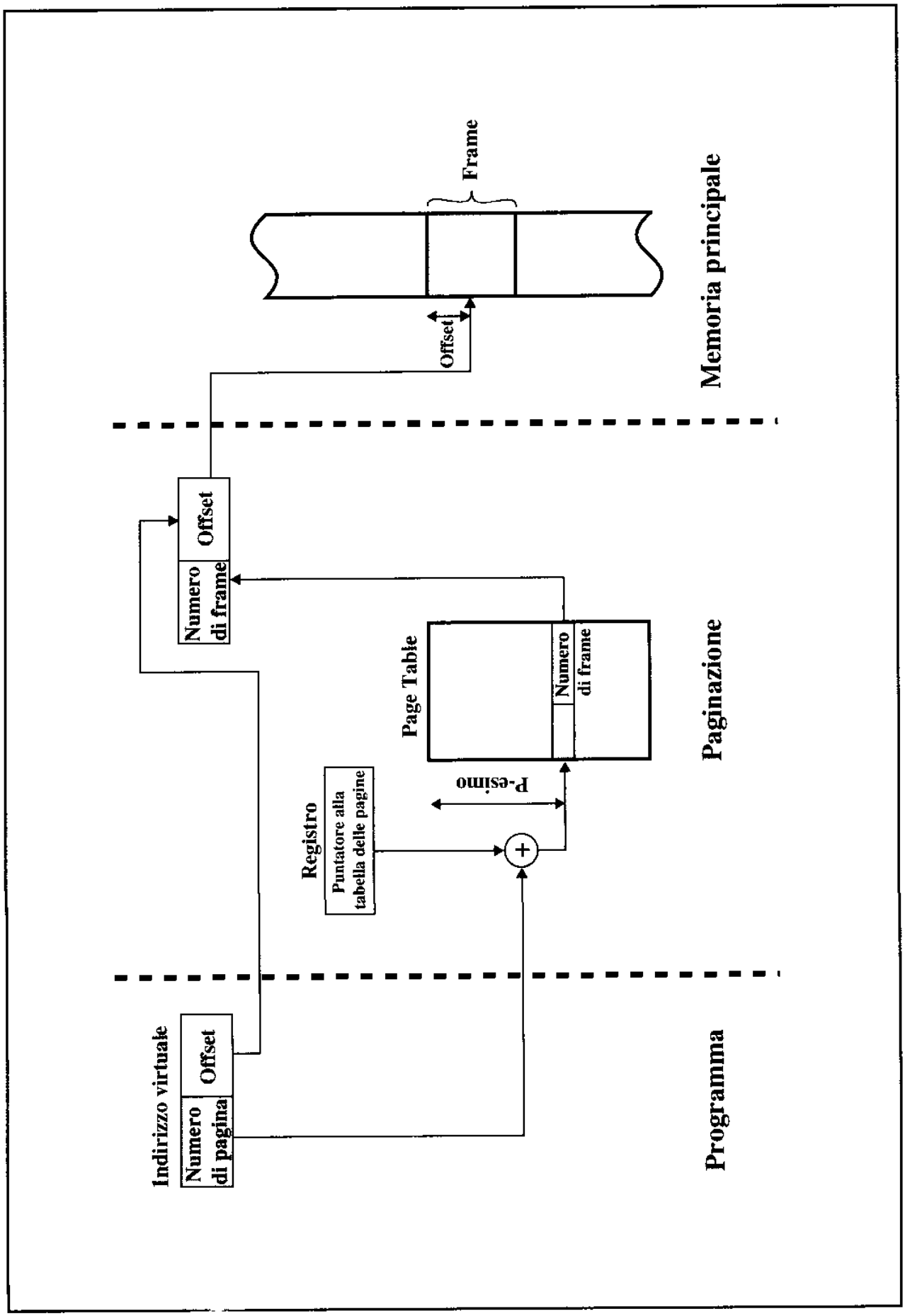


Figura 8.3 *Traduzione degli indirizzi in un sistema a paginazione*

za di una tabella delle pagine è uguale ad una pagina; ad esempio, il processore Pentium usa quest'approccio.

Un approccio alternativo all'uso di tabelle delle pagine ad uno o due livelli è l'uso di una struttura di tabella delle pagine invertita (**inverted page table**) (Figura 8.4): quest'approccio è usato sul PowerPC e sull'AS/400 dell'IBM. Anche un'implementazione del sistema operativo Mach su RT-PC usa questa tecnica.

In quest'approccio, la parte contenente il numero della pagina di un indirizzo virtuale è mappato in una tavola hash usando una semplice funzione di hashing[1]. La tavola hash contiene un puntatore alla tabella delle pagine invertita, che contiene le entry della tabella delle pagine. Con questa struttura, c'è solo una entry nella tavola hash e una nella tabella delle pagine invertita, per ogni pagina di memoria reale, invece che una per pagina virtuale: perciò una porzione fissa della memoria reale è occupata dalla tavola qualunque sia il numero dei processi o delle loro pagine virtuali nel sistema. Poiché più di un indirizzo virtuale può corrispondere alla stessa tavola hash, si usa una tecnica a lista per gestire l'overflow: osserviamo che la tecnica di hash costruisce generalmente liste corte, di uno o due elementi.

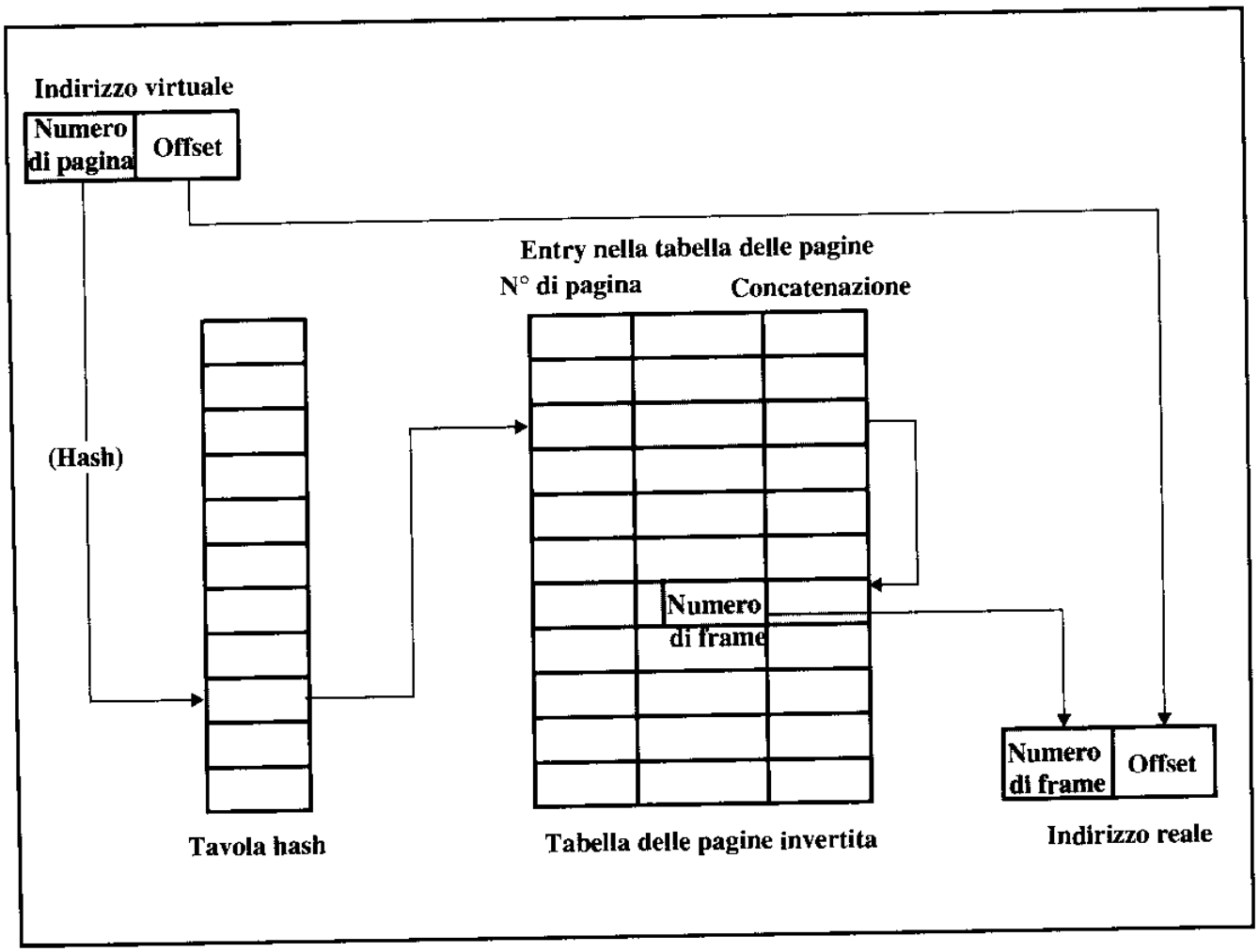


Figura 8.4 *Struttura di una tabella delle pagine invertita*

[1] Vedere l'Appendice 8A per una discussione sull'uso di tecniche hash.

Translation Lookaside Buffer

In linea di principio, ogni riferimento a memoria virtuale può causare due accessi alla memoria fisica: uno per andare a prendere l'entry della tabella delle pagine e uno per andare a prendere il dato desiderato. Perciò, un semplice schema di memoria virtuale avrebbe l'effetto di raddoppiare il tempo di accesso alla memoria. Per superare questo problema, la maggior parte degli schemi di memoria virtuale fanno uso di una speciale cache per le entry della tabella delle pagine, solitamente chiamata **translation lookaside buffer**(TLB). Questa cache funziona nello stesso modo della cache di memoria (Capitolo 1) e contiene quelle entry della tabella delle pagine che sono state usate più recentemente. L'organizzazione dell'hardware di paginazione risultante è illustrata nella Figura 8.5. Dato un indirizzo virtuale, il processore per prima cosa esaminerà il TLB. Se l'entry alla tabella delle pagine desiderata è presente (successo di TLB), allora si recupera il numero di frame e si forma l'indirizzo reale; se l'entry alla tabella delle pagine non è presente (insuccesso di TLB), allora il processore usa il numero di pagina come indice nella tabella delle pagine, per esaminare la corrispondente entry della tabella delle pagine. Se il "present bit" ha valore 1, allora la pagina è nella memoria principale, e il processore può recuperare il numero di frame dall'entry della tabella delle pagine per formare l'indirizzo reale; il processore aggiorna anche il TLB includendo questa nuova entry della tabella delle pagine. Infine, se il present bit vale 0, allora la pagina desiderata non è nella memoria principale: avviene un **fault di pagina**, cioè il fallimento dell'accesso alla memoria. A questo punto, lasciamo il regno dell'hardware per attivare il sistema operativo, che carica la pagina necessaria e aggiorna la tabella delle pagine.

La Figura 8.6 è un diagramma di flusso che mostra l'uso del TLB. Il diagramma mostra che se la pagina desiderata non è nella memoria principale, un'interruzione di fault di pagina provoca la chiamata della routine che gestisce l'interrupt di fault di pagina. Per mantenere semplice il diagramma di flusso, non mostriamo il fatto che il sistema operativo può avviare un altro processo, mentre l'I/O su disco è in corso. Per il principio di località, la maggior parte dei riferimenti a memoria virtuale saranno a locazioni di pagine usate recentemente, perciò la maggior parte dei riferimenti coinvolgerà le entry della tabella delle pagine già nella cache. Studi del TLB del VAX hanno dimostrato che questo schema può migliorare significativamente la prestazioni.

C'è un certo numero di dettagli aggiuntivi a proposito dell'organizzazione pratica del TLB. Poiché il TLB contiene solo alcune entry nell'intera tabella delle pagine, non possiamo semplicemente usare un indice nel TLB dato dal numero di pagina: occorre che ogni entry nel TLB contenga sia il numero della pagina sia l'entry completa della tabella delle pagine. Il processore è fornito di hardware che gli consente di interrogare simultaneamente tutte le entry del TLB per determinare se c'è una corrispondenza con il numero di pagina. Questa tecnica è detta mappa associativa, ed è confrontata in Figura 8.7 con la mappa diretta, o indicizzazione, usata per cercare nella tabella delle pagine. La progettazione del TLB deve anche considerare il modo in cui le entry sono organizzate nel TLB, e quali entry devono essere sostituite quando è introdotta una nuova entry: tali questioni devono essere considerate nella progettazione di qualunque cache di hardware. Quest'argomento non è trattato qui; il lettore può consultare documentazione specifica sulla progettazione della cache per ulteriori dettagli (ad esempio [STAL96]).

Infine, il meccanismo di memoria virtuale deve interagire con il sistema di cache (non la cache di TLB, ma la cache della memoria principale). Questo è illustrato nella Figura 8.8: un

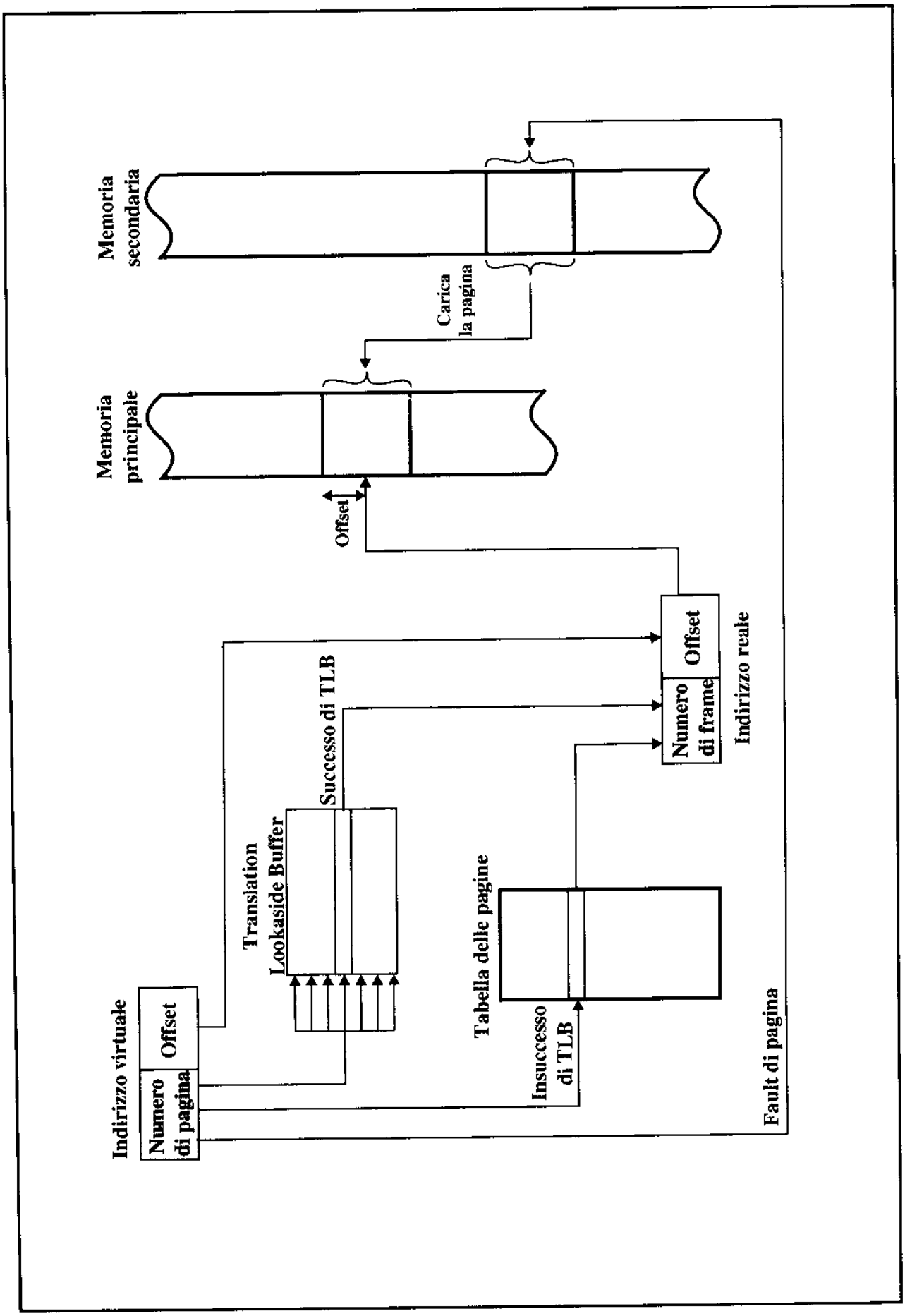


Figura 8.5 *Uso del translation lookaside buffer*

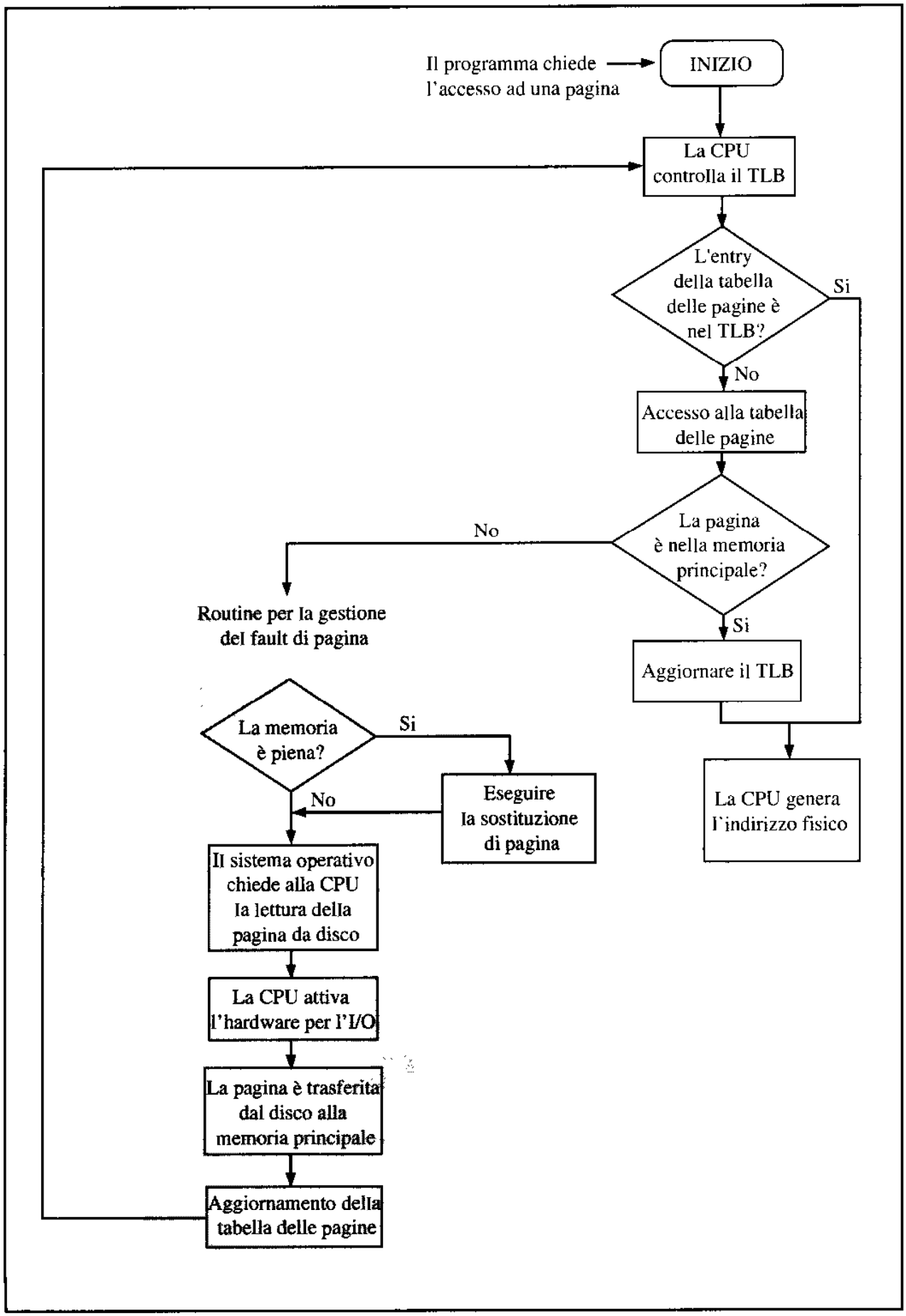


Figura 8.6 *Operazione di paginazione e Translation Lookaside Buffer [FURH87]*

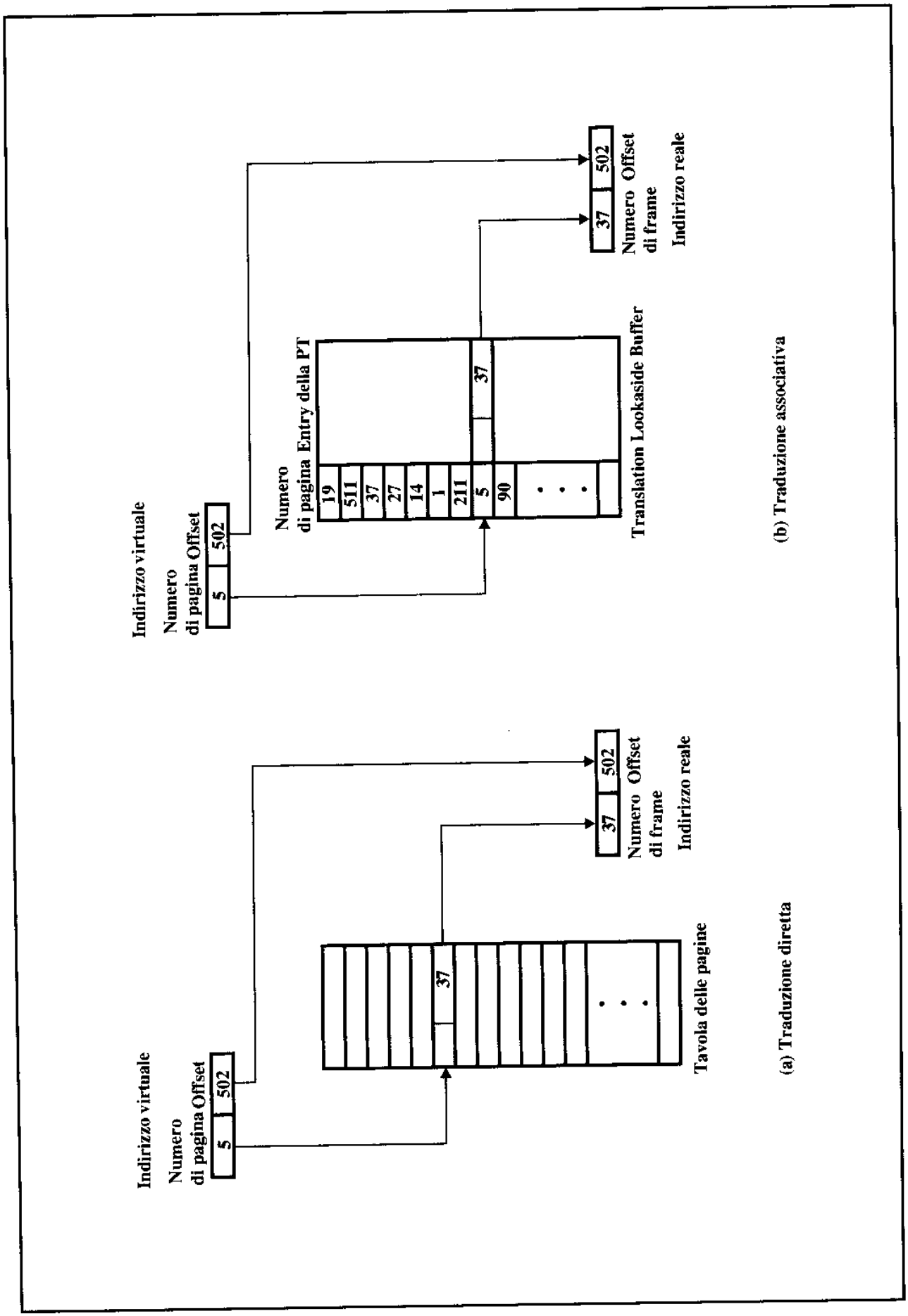


Figura 8.7 *Confronto fra ricerca diretta e ricerca associativa per le entry nella tabella delle pagine*

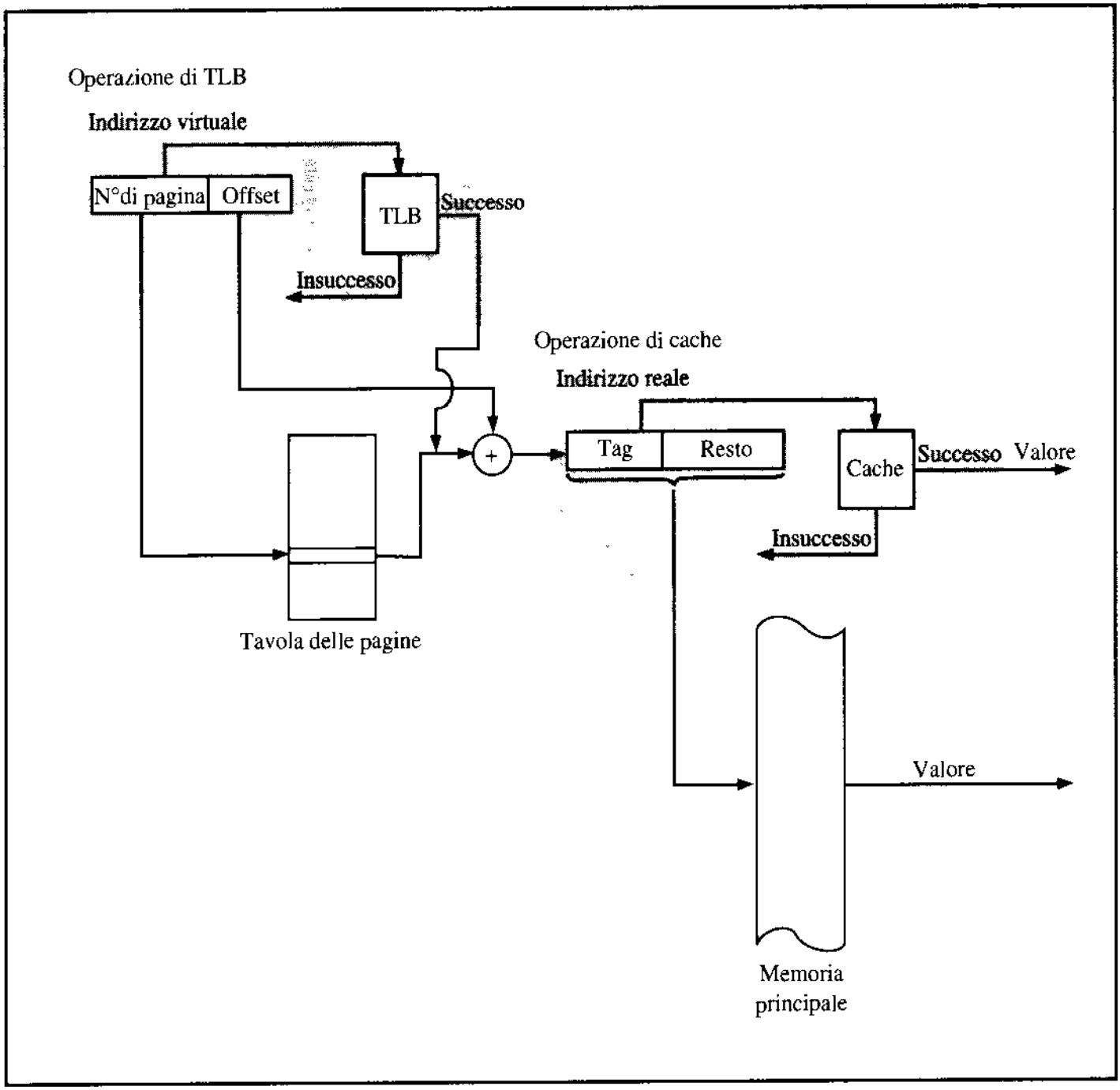


Figure 8.8 *Operazioni di cache e Translation Lookaside Buffer*

indirizzo virtuale avrà la forma di (numero di pagina, offset). In primo luogo, il sistema di gestione della memoria consulta il TLB per vedere se vi è presente la corrispondente entry nella tabella delle pagine: se è così, l'indirizzo reale (fisico) si genera combinando il numero di frame con l'offset. Se non è presente, si accede all'entry della tabella delle pagine. Una volta che l'indirizzo reale è stato generato, in forma di tag[2] e resto, si consulta la cache per vedere se il blocco contenente la parola è presente. Se è così, la parola è restituita alla CPU; altrimenti si recupera la parola dalla memoria principale.

[2] Vedere Figura 1.17. Tipicamente, un tag è formato dai bit più a sinistra di un indirizzo reale. Per una dettagliata trattazione sulle cache, consultare [STAL96]

Il lettore dovrebbe essere in grado di apprezzare la complessità dell'hardware della CPU coinvolto in un singolo riferimento a memoria. L'indirizzo virtuale è tradotto in indirizzo reale, coinvolgendo il riferimento ad una entry della tabella delle pagine, che può essere nel TLB, nella memoria principale, o su disco. La parola cui ci si riferisce può essere nella cache, nella memoria principale o sul disco: nell'ultimo caso, la pagina contenete la parola deve essere caricata nella memoria principale, il suo blocco deve essere caricato nella cache e, inoltre, l'entry corrispondente della tabella delle pagine deve essere aggiornata.

Dimensione della pagina

Un'importante decisione nella progettazione dell'hardware è la dimensione di pagina che deve essere usata. Ci sono diversi fattori da considerare, cominciando dalla frammentazione interna: chiaramente, più piccola è la dimensione della pagina, minore è la frammentazione interna, che deve essere ridotta per ottimizzare l'uso della memoria principale. D'altra parte, più piccola è la pagina, maggiore è il numero delle pagine richieste per processo; e più pagine per processo significano tabella delle pagine più grande. Per i grandi programmi in un ambiente pesantemente multiprogrammato, questo può significare che alcune porzioni della tabella delle pagine dei processi attivi devono essere nella memoria virtuale e non nella memoria principale. Perciò ci può essere un doppio fault di pagina per un singolo riferimento a memoria: il primo per caricare la porzione desiderata della tabella delle pagine, il secondo per caricare la pagina del processo. Un altro fattore è che le caratteristiche fisiche della maggior parte dei dispositivi per la memoria secondaria favoriscono una dimensione di pagina più grande, per un più efficiente trasferimento dei blocchi di dati.

La dimensione della pagina complica questi problemi, perché influenza la percentuale di occorrenze di fault di pagina: questo comportamento, in termini generali, è illustrato in Figura 8.9a, ed è basato sul principio di località. Se la dimensione della pagina è molto piccola, allora solitamente sarà disponibile un gran numero di pagine nella memoria principale per un processo. Dopo un certo tempo, le pagine in memoria conterranno parti dei processi vicini agli accessi recenti. Perciò, la percentuale di occorrenza di fault di pagina dovrebbe essere bassa. Al crescere della dimensione della pagina, ogni singola pagina conterrà locazioni sempre più distanti dai riferimenti recenti, perciò l'effetto del principio di località è indebolito e la percentuale di fault di pagina incomincia a crescere. Prima o poi, comunque, la frequenza di fault di pagina comincerà a scendere, non appena la dimensione della pagina si avvicinerà alla dimensione dell'intero processo (punto P nel diagramma). Quando una singola pagina contiene l'intero processo, non ci saranno più fault di pagina.

Il fatto che la percentuale del fault di pagina sia determinata anche dal numero di frame allocati per processo complica la situazione. La Figura 8.9b mostra che, fissando la dimensione di pagina, la percentuale di fault diminuisce, non appena il numero di pagine mantenute nella memoria principale cresce[3]. Perciò, una politica software (la quantità di memoria da allocare per ogni processo) influenza una decisione di progettazione hardware.

---

[3] Il parametro W rappresenta la dimensione del working set, un concetto che vedremo nella sezione 8.2.

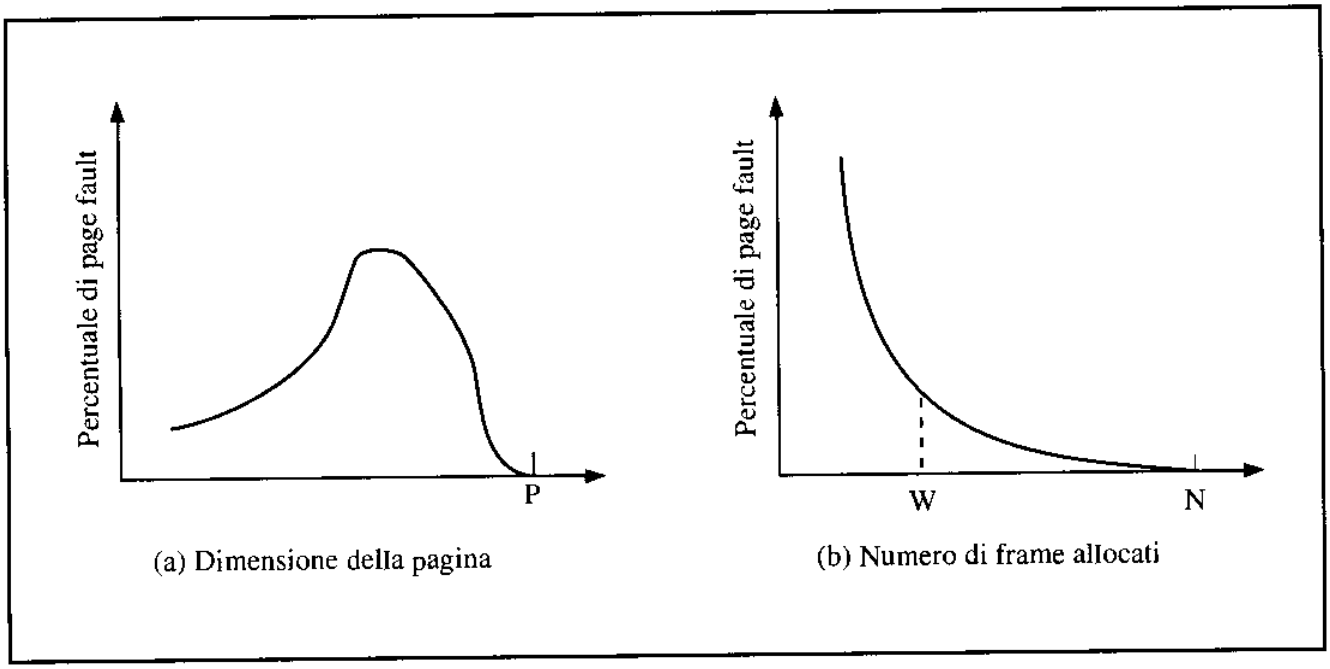


Figura 8.9 *Comportamento tipico di un programma durante la paginazione*

La Tabella 8.2 indica le dimensioni delle pagine usate su alcune macchine.

Infine, la questione della progettazione della dimensione delle pagine è in relazione con la dimensione della memoria fisica principale. Le implementazioni di memoria virtuale nella maggior parte dei sistemi odierni sono state progettate nell'ipotesi che la memoria fisica non fosse troppo grande, tipicamente tra i 4MB e i 256MB. Comunque, ci possiamo aspettare di vedere dimensioni di memoria fisica più grandi, su workstation e macchine più grandi, nei prossimi anni.

Nel momento in cui cresce la memoria principale. aumenta anche lo spazio degli indirizzi usato dalle applicazioni. La tendenza è più ovvia sui personal computer e sulle workstation, dove le applicazioni stanno diventando sempre più complesse. Inoltre, le tecniche contemporanee di programmazione usate per grandi programmi tendono a diminuire la località dei riferimenti all'interno di un processo [HUCK93]. Per esempio:

Tabella 8.2 *Esempi di dimensioni di pagina*

Computer	Dimensione della pagina
Atlas	512 word di 48 bit
Honeywell-Multics	1024 word di 36 bit
IBM 370/XA e 370/ESA	4 kilobyte
Famiglia dei VAX	512 byte
IBM AS/400	512 byte
DEC Alpha	8 kilobyte
Pentium	4 kilobyte o 4 megabyte
Power PC	4 kilobyte

- Le tecniche orientate agli oggetti incoraggiano l'uso di molti piccoli programmi e moduli di dati con riferimenti sparsi su un numero relativamente alto di oggetti, in un periodo relativamente breve.
- Le applicazioni multithread provocano repentini cambiamenti nel flusso di istruzioni e riferimenti di memoria sparsi.

Per una data dimensione di translation lookaside buffer (TLB), più la dimensione di memoria dei processi cresce, più la località decresce, più le possibilità di successo nell'accesso al TLB decrescono; in queste circostanze, il TLB può diventare un collo di bottiglia per le prestazioni (vedere ad esempio [CHEN92]).

Un modo per migliorare le prestazioni del TLB è usare un TLB più grande con più entry. Comunque, la dimensione del TLB interagisce con gli altri aspetti della progettazione di hardware, come la cache della memoria principale e il numero di accessi a memoria per ciclo di istruzioni [TALL92]; ma la dimensione del TLB non cresce rapidamente quanto la dimensione della memoria principale. Un'alternativa è usare dimensioni di pagina più grandi, in modo che ogni entry alla tabella delle pagine nel TLB si riferisca a blocchi di memoria più grandi; ma abbiamo appena visto che l'uso di pagine di grandi dimensioni può portare ad un peggioramento delle prestazioni.

Quindi, alcuni progettisti hanno studiato l'uso di più dimensioni di pagina [TALL92, KHAL93], e diverse architetture di microprocessori le supportano, tra cui R4000, Alpha, SuperSPARC e Pentium. Per esempio, R4000 supporta sette diverse dimensioni di pagina (da 4KB a 16MB). L'uso di pagine con dimensioni diverse consente la flessibilità necessaria per usare efficacemente il TLB; per esempio, grandi regioni contigue nello spazio di indirizzi di un processo, come le istruzioni dei programmi, possono essere mappate in un piccolo numero di grandi pagine, piuttosto che in un grande numero di piccole pagine, mentre gli stack dei thread possono essere mappati usando pagine di piccola dimensione.

La segmentazione

Implicazioni della memoria virtuale

La segmentazione permette al programmatore di vedere la memoria come un insieme di segmenti o spazi di indirizzamento multipli. I segmenti possono essere di dimensione diversa, anzi dinamica; i riferimenti a memoria sono indirizzi della forma (numero di segmento, offset).

Questa organizzazione offre dei vantaggi al programmatore, rispetto ad uno spazio di indirizzi non segmentato:

1. Semplifica la gestione di strutture dati che crescono. Se il programmatore non conosce in anticipo quanto diventerà grande una struttura dati, è necessario tirare ad indovinarlo, a meno che non sia permesso usare segmenti di dimensioni dinamiche. Con la memoria virtuale segmentata, si può assegnare un segmento alla struttura dati, e il sistema operativo espanderà o restringerà il segmento secondo i casi. Se un segmento, che necessita di essere esteso, è in memoria principale, e non c'è spazio a sufficienza per l'estensione, il sistema operativo può spostare il segmento in una regione più ampia della memoria principale, se disponibile, o

spostarlo su disco. In quest'ultimo caso, il segmento allargato sarà ricaricato in memoria alla prossima opportunità.

2. Permette la modifica e la ricompilazione indipendente dei programmi, senza richiedere che si rifaccia il link con tutto l'insieme dei programmi per ricaricarli: anche questo è realizzabile usando segmenti multipli.
3. Si presta alla condivisione tra processi: un programmatore può allocare un programma di utilità o un'utile tabella di dati in un segmento, che può essere accessibile agli altri processi.
4. Si presta alla protezione: poiché un segmento può contenere un insieme ben definito di programmi o dati, il programmatore o l'amministratore di sistema può assegnare privilegi in modo conveniente.

Organizzazione

Nella discussione sulla segmentazione semplice, abbiamo indicato che ogni processo ha la sua tabella dei segmenti e, quando tutti i suoi segmenti sono caricati nella memoria principale, la tabella dei segmenti per tale processo è creata e caricata nella memoria principale. Ogni entry della tabella dei segmenti contiene l'indirizzo di partenza del segmento corrispondente nella memoria principale e la lunghezza del segmento. Lo stesso strumento, una tabella di segmenti, è necessario quando consideriamo uno schema di memoria virtuale basato sulla segmentazione: anche in questo caso, è tipico associare un'unica tabella dei segmenti ad ogni processo, ma le entry della tabella dei segmenti diventano più complesse (Figura 8.2b). Poiché solo alcuni dei segmenti di un processo possono essere nella memoria principale, è necessario un bit per ogni entry della tabella dei segmenti per indicare se il segmento corrispondente è presente o meno nella memoria principale. Se il bit indica che il segmento è in memoria, allora l'entry contiene anche l'indirizzo di partenza e la lunghezza di quel segmento.

Un altro bit di controllo nella entry della tabella dei segmenti è il modify bit, che indica se i contenuti del segmento corrispondente sono stati alterati dopo l'ultimo caricamento in memoria: se non ci sono stati cambiamenti, allora non è necessario scrivere il segmento su disco quando è il momento di sostituirlo nel frame che occupa attualmente. Possono essere presenti altri bit di controllo; per esempio, se la protezione o la condivisione sono gestite a livello di segmenti, allora saranno necessari alcuni bit per tale scopo.

Il meccanismo di base per leggere una parola dalla memoria richiede la traduzione di un indirizzo virtuale o logico, formato dal numero di segmento e dall'offset, in un indirizzo fisico, usando una tabella dei segmenti. Poiché la tabella dei segmenti è di lunghezza variabile, secondo la dimensione del processo, non si può pensare di mantenerla in registri: deve essere, infatti, nella memoria principale perché vi si possa accedere. La Figura 8.10 suggerisce un'implementazione hardware di questo schema: quando un particolare processo è in esecuzione, un registro mantiene l'indirizzo di partenza della tabella dei segmenti per quel processo. Il numero di segmento di un indirizzo virtuale è usato come indice in quella tabella, per cercare l'indirizzo di memoria principale a cui inizia il segmento; quest'ultimo è aggiunto all'offset dell'indirizzo virtuale per produrre l'indirizzo reale desiderato.

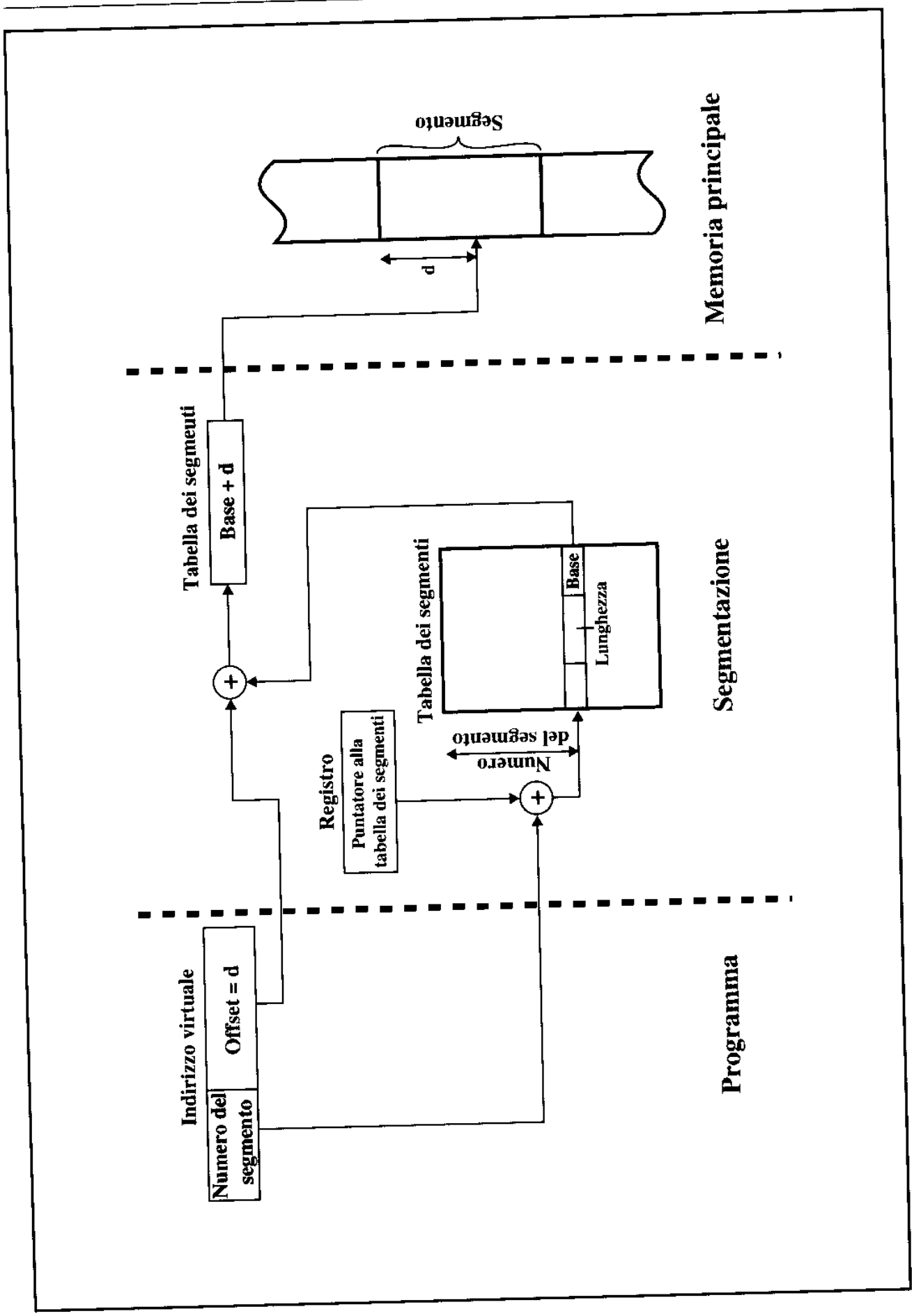


Figura 8.10 *Traduzione di indirizzo in un sistema a segmentazione*

Combinare paginazione e segmentazione

Sia la paginazione sia la segmentazione hanno i propri punti forti: la paginazione, trasparente al programmatore, elimina il problema della frammentazione interna e consente quindi un uso efficiente della memoria principale. Inoltre, poiché i pezzi caricati nella memoria principale e i pezzi scaricati su disco hanno uguale dimensione fissa, è possibile sviluppare (come vedremo) sofisticati algoritmi per la gestione della memoria sfruttando il comportamento dei programmi. La segmentazione, visibile al programmatore, ha i vantaggi elencati in precedenza, tra cui la capacità di gestire strutture dati dinamiche, la modularità e il supporto per la condivisione e la protezione. Per combinare i vantaggi di entrambe, alcuni sistemi sono dotati di processori hardware e di software di sistema operativo che le forniscono tutte e due.

In un sistema che combina paginazione e segmentazione, uno spazio di indirizzo utente è diviso in un certo numero di segmenti, a discrezione del programmatore; ogni segmento è a sua volta suddiviso in pagine di dimensione fissa, lunghe quanto i frame della memoria principale. Se un segmento è più corto di una pagina, occupa solo una pagina. Dal punto di vista del programmatore, un indirizzo logico è ancora formato da numero del segmento e offset, mentre dal punto di vista del sistema, l'offset del segmento è visto come numero di pagina e offset di pagina all'interno del segmento specificato.

La Figura 8.11 suggerisce una struttura per combinare paginazione e segmentazione: ogni processo ha una tabella dei segmenti e, per ogni segmento, una tabella delle pagine. Durante l'esecuzione del processo, un registro contiene l'indirizzo di partenza della tabella dei segmenti di quel processo; in presenza di un indirizzo virtuale, il processore usa il numero di segmento come indice nella tabella dei segmenti, per trovare la tabella delle pagine del segmento in questione. Quindi si usa il numero di pagina, contenuto nell'indirizzo virtuale, come indice nella tabella delle pagine, per cercare il numero del frame ad essa corrispondente che, combinato con l'offset dell'indirizzo virtuale, produce l'indirizzo reale desiderato.

La Figura 8.2c suggerisce i formati delle entry della tabella dei segmenti e della tabella delle pagine: come prima, la entry della tabella dei segmenti contiene la lunghezza del segmento ed un campo base, che si riferisce alla tabella delle pagine. Il presence bit e il modify bit non sono necessari, perché gestiti a livello di pagina; altri bit di controllo possono essere usati per scopi di condivisione e protezione. La entry della tabella delle pagine è essenzialmente la stessa usata in un sistema a paginazione pura; ogni numero di pagina è associato al numero di frame corrispondente, se la pagina è presente nella memoria principale, e il modify bit indica se questa pagina deve essere riscritta nuovamente su disco quando si alloca il frame per un'altra pagina. Ci possono essere altri bit di controllo per la protezione e per altri aspetti della gestione della memoria.

Protezione e condivisione

La segmentazione si presta all'implementazione delle tecniche di protezione e condivisione. Poiché la entry di ogni tabella dei segmenti contiene una lunghezza e un indirizzo di base, un programma non può inavvertitamente accedere ad una locazione di memoria principale il cui indirizzo superi l'indirizzo limite del segmento. Per ottenere la condivisione, è possibile avere un riferimento allo stesso segmento nella tabella dei segmenti di più di un processo. Gli stessi meccanismi sono naturalmente disponibili anche in un sistema a paginazione; in quest'ultimo

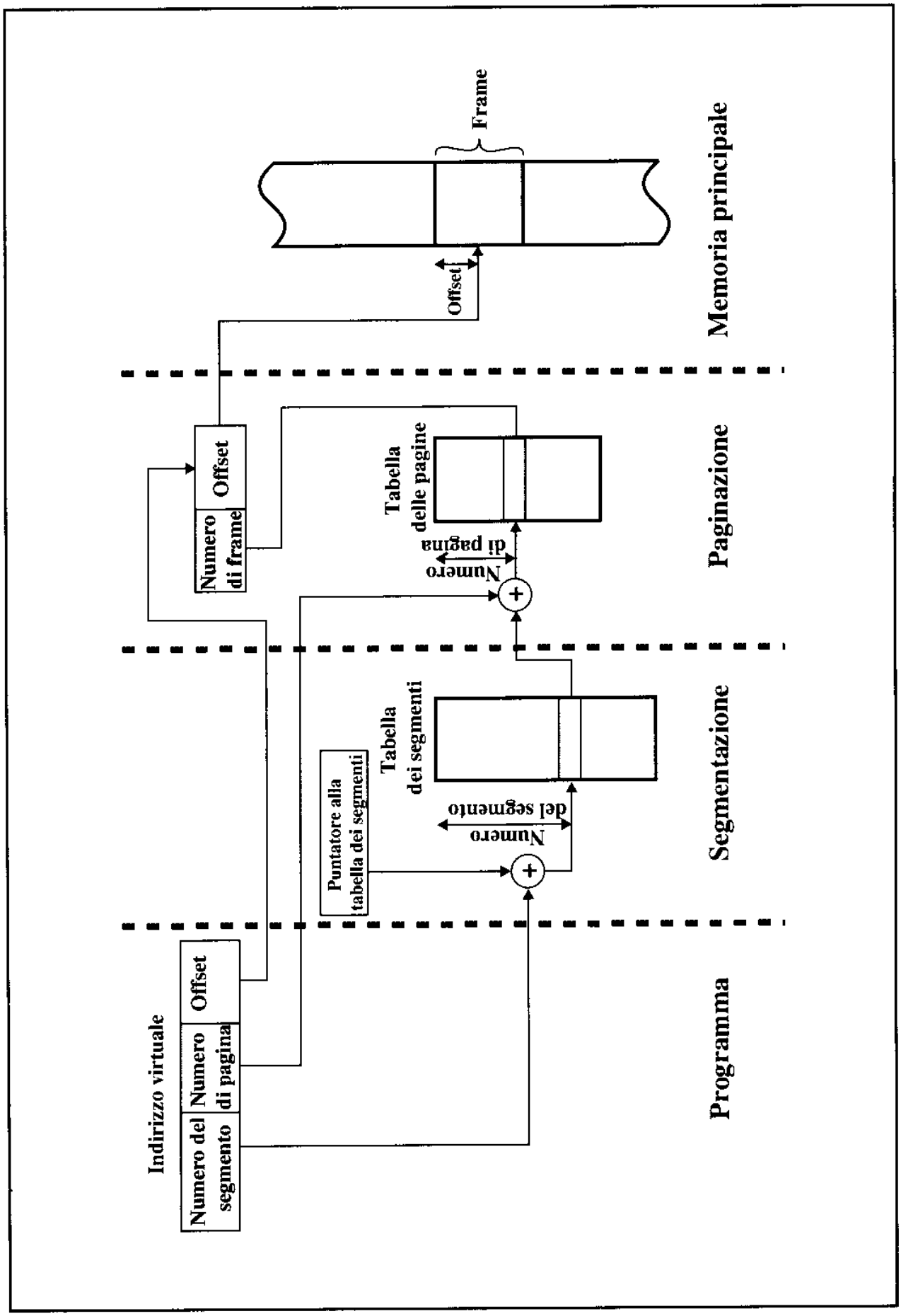


Figura 8.11 *Traduzione di indirizzo in un sistema a segmentazione e paginazione combinate*

caso, la struttura delle pagine dei programmi e dei dati non è visibile al programmatore, rendendo le specifiche di protezione e i requisiti di condivisione più difficili. La Figura 8.12 illustra i tipi di relazioni di protezione che possono essere assicurate in tale sistema.

Possono essere anche forniti meccanismi più sofisticati. Uno schema comune è usare una struttura di protezione ad anello del tipo di cui si è parlato nel capitolo 3 (Problema 3.7). In questo schema, anelli con numero più basso o più interni godono di privilegi maggiori di anelli con numero più alto o più esterni. Tipicamente, l'anello 0 è riservato per le funzioni del kernel del sistema operativo, con le applicazioni ai livelli superiori; alcuni programmi di utilità o servizi del sistema operativo possono occupare un anello intermedio. I principi basilari di un sistema ad anello sono i seguenti:

1. Un programma può accedere solo a dati che risiedono sullo stesso anello o su un anello con meno privilegi.
2. Un programma può chiamare servizi che risiedono sullo stesso anello o su un anello più privilegiato.

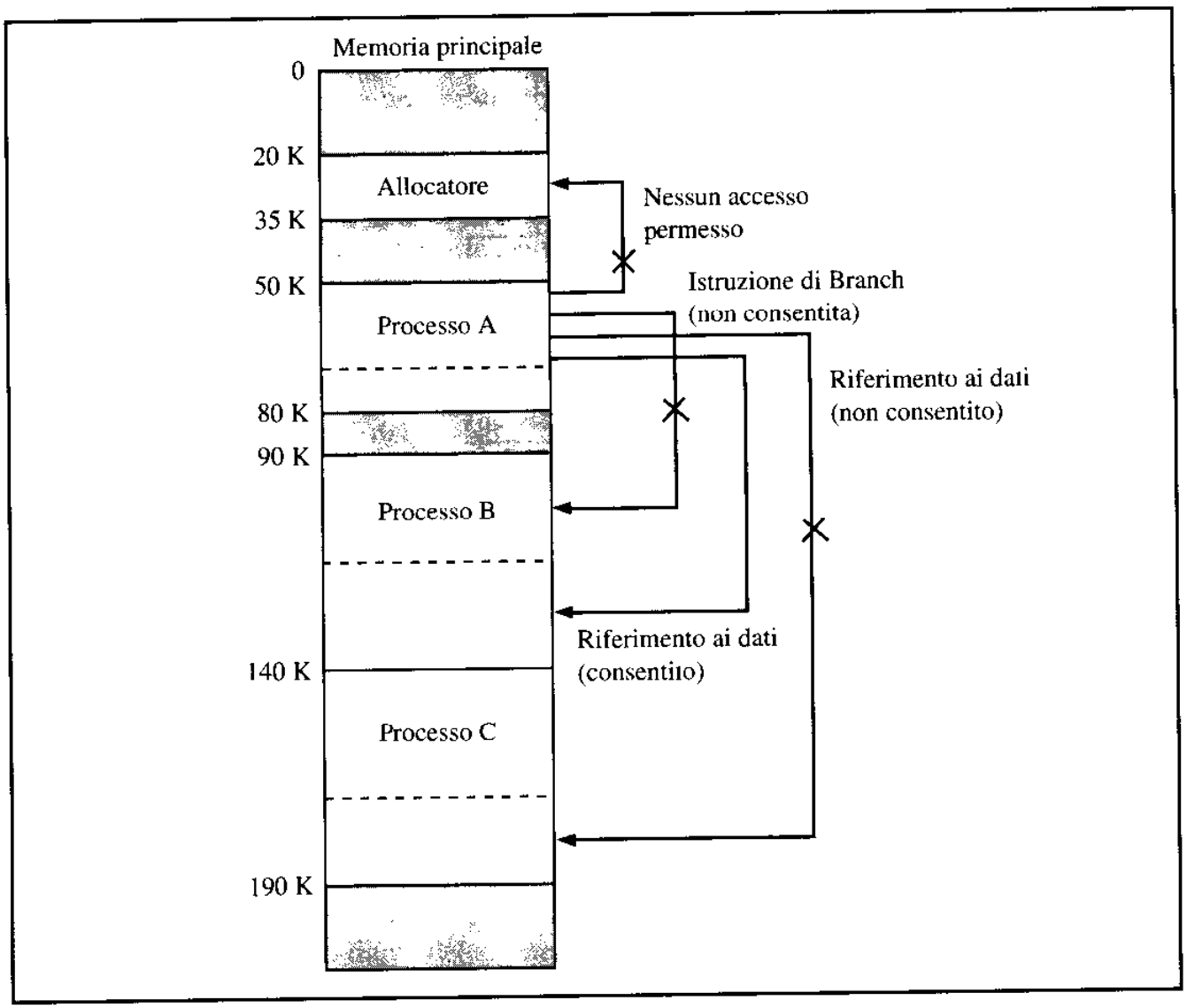


Figura 8.12 *Relazioni di protezione tra i segmenti*

8.2 Il software del sistema operativo

La progettazione della parte del sistema operativo riguardante la gestione della memoria dipende da tre aree di scelta fondamentali:

- Usare o meno le tecniche di memoria virtuale.
- L'uso della paginazione, o della segmentazione, o di entrambe
- Algoritmi utilizzati per la gestione della memoria.

Le scelte fatte nelle prime due aree dipendono dalla piattaforma hardware disponibile. Perciò, le prime implementazioni di UNIX non fornivano memoria virtuale perché i processori sui quali il sistema girava non supportavano la paginazione o la segmentazione. Nessuna di queste due tecniche è pratica senza il supporto hardware per la traduzione di indirizzi e per altre funzioni basilari.

È utile fare due commenti sui primi due punti menzionati sopra. In primo luogo, ad eccezione dei sistemi operativi di alcuni vecchi PC, come MS-DOS, e sistemi specializzati, tutti i sistemi operativi importanti sono dotati di memoria virtuale; in secondo luogo, i sistemi a pura segmentazione stanno diventando sempre più rari. Usando la combinazione di paginazione e segmentazione, la maggior parte dei problemi riguardanti la gestione della memoria che un progettista di sistemi operativi si trova ad affrontare riguardano la paginazione[4]. Perciò in questa sezione saranno trattate le questioni connesse con la paginazione.

Le scelte fatte nella terza area sono di competenza del software di sistema operativo e saranno l'argomento di questa sezione. La Tabella 8.3 elenca gli elementi chiave della progettazione che devono essere esaminati. In ogni caso, la questione chiave riguarda le prestazioni: si vuole minimizzare la frequenza di fault di pagina, che causano un considerevole sovraccarico software.

Tabella 8.3 *Strategie del sistema operativo per la gestione della memoria virtuale*

Strategia di fetch	**Gestione del resident set**
A richiesta	Dimensione del residenı set
Prepaginazione	Fissata
	Variabile
Strategia di posizionamento	Ambito di sostituzione
	Globale
Strategia di sostituzione	Locale
Algoritmi di base	**Strategia di cleaning**
Ottimo	A richiesta
Least Recently Used (LRU)	Precleaning
First in, First Out (FIFO)	
Orologio	**Controllo del carico**
Buffer per pagine	Grado di multiprogrammazione

[4] Protezione e condivisione sono trattate a livello di segmento, nei sistemi con paginazione e segmentazione combinate; tratteremo di questo argomento nei capitoli successivi.

Al minimo, il sistema operativo deve decidere quali pagine residenti deve sostituire, al costo dell'I/O per lo scambio di pagine; quindi deve schedulare un altro processo da eseguire durante le operazioni di I/O sulla pagina, causando un cambio di processo; di conseguenza, bisognerebbe fare in modo che, durante l'esecuzione di un programma, la probabilità di riferirsi ad una parola in un pagina mancante, sia minima. In tutti i campi a cui ci si riferisce nella Tabella 8.3, non esiste una strategia che lavora al meglio. Come si può vedere, il compito della gestione della memoria in un ambiente di paginazione è terribilmente complesso. Inoltre, le prestazioni di ogni insieme di strategie, dipendono dalla dimensione della memoria principale, dalla velocità relativa della memoria principale e secondaria, dalla dimensione e dal numero di processi che si contendono le risorse, e dal comportamento in esecuzione dei singoli programmi. Quest'ultima caratteristica dipende a sua volta dalla natura dell'applicazione, dal linguaggio di programmazione e dal compilatore utilizzati, dallo stile del programmatore e, per un programma interattivo, dal comportamento dell'utente; perciò, il lettore non può aspettarsi risposte definitive né qui né in altri testi. Per sistemi più piccoli, il progettista del sistema operativo può tentare di scegliere un insieme di strategie che sembrano "buone" in un gran numero di condizioni, basate sullo stato attuale delle conoscenze; per sistemi più grandi, in particolare per i mainframe, il sistema operativo deve essere fornito di strumenti di controllo, che consentono al gestore dell'installazione di mettere a punto il sistema operativo per ottenere "buoni" risultati basandosi sulle condizioni locali.

Strategia di fetch

La strategia di fetch determina quando una pagina deve essere caricata nella memoria principale. Le due alternative comuni sono: paginazione a richiesta e prepaginazione. Con la **paginazione a richiesta** (*demand paging*), una pagina è caricata in memoria solo quando si fa un riferimento ad una locazione su quella pagina. Se gli altri elementi della strategia di gestione della memoria sono buoni, può succedere quanto segue. Quando un processo è avviato, si verificheranno moltissimi fault di pagina, e a mano a mano che le pagine sono caricate, il principio di località suggerisce che la maggior parte dei riferimenti futuri saranno a pagine che sono state caricate recentemente; perciò, dopo un certo tempo, il sistema si assesterà e il numero di fault di pagina decrescerà drasticamente.

Con la **prepaginazione** (*prepaging*), altre pagine (oltre a quella che ha generato il fault di pagina) sono caricate: l'attrattiva di questa tecnica è dovuta alla caratteristica della maggior parte dei dispositivi di memoria secondaria, come i dischi, che hanno tempi di posizionamento e latenza di rotazione, quindi se le pagine di un processo sono memorizzate in aree contigue della memoria secondaria, è più efficiente caricare un certo numero di pagine contigue tutte in una volta, piuttosto che caricarle una alla volta per un lungo periodo. Naturalmente, questa strategia è inefficiente se poi non si fa riferimento alla maggior parte delle pagine caricate in più.

La strategia di prepaginazione potrebbe essere utilizzata sia quando il processo è avviato per la prima volta, nel qual caso il programmatore dovrebbe indicare le pagine desiderate, sia ogni qualvolta avvenga un fault di pagina. Quest'ultima condotta dovrebbe essere preferibile perché è invisibile al programmatore; in ogni caso, l'utilità della prepaginazione non è stata stabilita [MAEK87].

La prepaginazione non deve essere confusa con il trasferimento di un processo su disco, o swapping: quando un processo è scaricato su disco e quindi sospeso, tutte le sue pagine residenti sono rimosse dalla memoria principale, quando si riattiva il processo, tutte le pagine che erano precedentemente nella memoria principale vi sono riportate. La maggior parte dei sistemi operativi segue questa strategia.

Strategia di posizionamento

La strategia di posizionamento determina dove deve risiedere un pezzo di processo nella memoria reale. In un sistema a pura segmentazione, la strategia di posizionamento è un importante problema di progettazione: le possibili alternative sono costituite da algoritmi come il best-fit, il first-fit e gli altri discussi nel Capitolo 7. Comunque per un sistema che usa o la paginazione pura oppure la paginazione combinata con la segmentazione, il posizionamento è solitamente irrilevante perché l'hardware di traduzione degli indirizzi e l'hardware per l'accesso alla memoria principale possono eseguire le loro funzioni per ogni combinazione pagina-frame con pari efficienza.

C'è un'area in cui il posizionamento può diventare una preoccupazione, e questo è un recente oggetto di ricerca e sviluppo. Su un multiprocessore con accesso a memoria non uniforme (NUMA, Non Uniform Memory Access), è possibile accedere alla memoria condivisa distribuita della macchina da qualunque processore, ma il tempo di accesso ad una particolare locazione fisica varia secondo la distanza del processore dal modulo di memoria. Perciò, le prestazioni dipendono fortemente da quanto i dati distano dal processore che li usa [LARO92, BOLO89, COX89]. Per i sistemi NUMA, è preferibile utilizzare una strategia di posizionamento automatico per assegnare le pagine al modulo di memoria che garantisce le migliori prestazioni.

Strategia di sostituzione

Nella maggior parte dei testi sui sistemi operativi, il trattamento della gestione della memoria contiene una sezione intitolata "strategia di sostituzione", che riguarda la selezione di una pagina in memoria da sostituire, quando una nuova pagina deve essere caricata. Questo argomento è talvolta di difficile spiegazione perché sono coinvolti diversi concetti collegati tra loro:

- Quanti frame devono essere allocati per ogni processo attivo.
- Se l'insieme delle pagine da considerare per la sostituzione deve essere limitato al numero di pagine del processo che causano fault di pagina, o deve contenere tutti i frame della memoria principale.
- Nell'insieme delle pagine considerate, quale pagina particolare deve essere selezionata per la sostituzione.

Ci si riferisce ai primi due concetti parlando di *resident set management* (gestione dell'insieme di pagine residenti), e questi saranno trattati nella prossima sottosezione, mentre si riserva il termine *strategia di sostituzione* per il terzo concetto, trattato in questa sezione.

L'area della strategia di sostituzione è stata probabilmente la più studiata nel campo della

gestione della memoria negli ultimi 20 anni. Quando tutti i frame della memoria principale sono occupati ed è necessario caricare una nuova pagina per soddisfare un fault di pagina, la strategia di sostituzione determina quale tra le pagine presenti in memoria deve essere sostituita. Tutte le strategie hanno come proprio obiettivo quello di rimuovere la pagina a cui ci si riferirà di meno nel prossimo futuro, ed a causa del principio di località, c'è spesso un'alta correlazione tra la più recente storia dei riferimenti e le sequenze di riferimenti future. Perciò, la maggior parte delle strategie cerca di predire il comportamento futuro, sulla base di quello passato. Un compromesso da considerare è che più è elaborata e sofisticata la tecnica di sostituzione, maggiore è il carico di hardware e software che la implementano.

Blocco di frame

Una restrizione alla strategia di sostituzione deve essere menzionata prima di considerare i diversi algoritmi: alcuni frame della memoria principale possono essere bloccati, e la pagina immagazzinata in quei frame non può essere sostituita. Buona parte del kernel del sistema operativo è conservata in frame bloccati, come le strutture chiave di controllo. Inoltre, i buffer di I/O e altre aree critiche per il tempo reale possono essere bloccati nei frame della memoria principale. È possibile bloccare i frame associando un lock bit ad ogni frame. Questo bit può essere nella tabella dei frame oppure nella tabella delle pagine.

Algoritmi di base

Oltre alla strategia di gestione del resident set (che sarà discussa nella prossima sottosezione), ci sono alcuni algoritmi di base che sono usati per selezionare la pagina da sostituire. Gli algoritmi di sostituzione che sono stati considerati in letteratura sono:

- Algoritmo ottimo (OPT)
- Least Recently Used (LRU)
- First-in, first-out (FIFO)
- Orologio.

L'**algoritmo ottimo** seleziona, per la sostituzione, la pagina alla quale ci si riferirà dopo il tempo più lungo. Si può dimostrare che questo algoritmo farà il numero più piccolo di fault di pagina. [BELA66]. Chiaramente, è impossibile implementare questo algoritmo, perché richiede che il sistema operativo abbia una perfetta conoscenza degli eventi futuri. Comunque, lo si può usare come termine di paragone per giudicare gli altri algoritmi.

La Figura 8.13 illustra un esempio dell'algoritmo ottimo. L'esempio suppone di avere per questo processo un'allocazione fissa di tre frame (dimensione fissa del resident set); l'esecuzione del processo richiede riferimenti a 5 pagine distinte, e la sequenza di indirizzamento a pagine che si forma eseguendo il programma è:

2 3 2 1 5 2 4 5 3 2 5 2

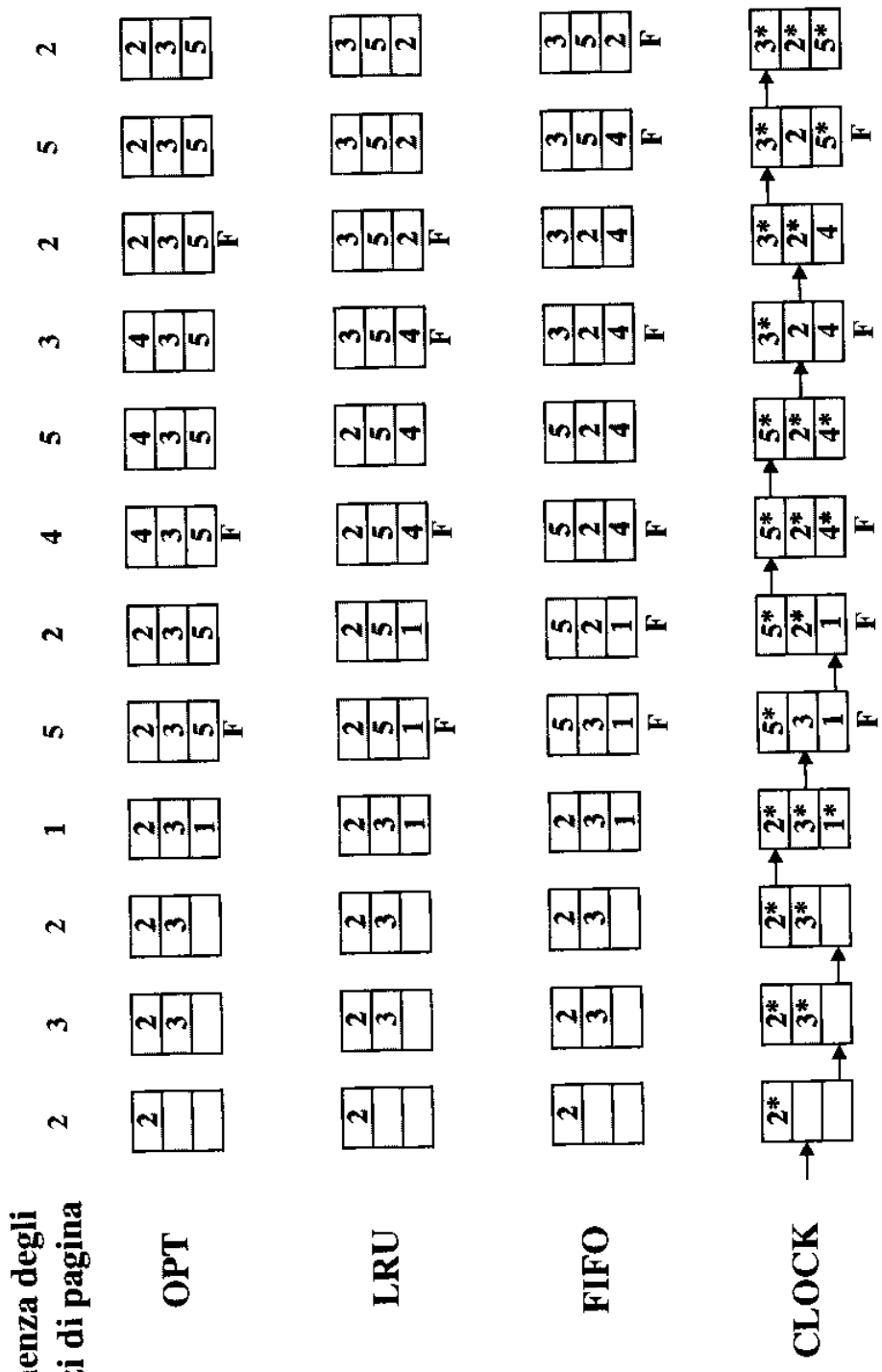


Figura 8.13 *Comportamento dei quattro algoritmi di sostituzione di pagine*

questo significa che la prima pagina a cui il processo fa riferimento è la 2, le seconda pagina è la 3 e così via. La strategia ottimale provoca 3 fault di pagina dopo che sono stati allocati tutti i frame.

La strategia di **least-recently-used**(LRU) sostituisce in memoria la pagina a cui il processo non si è riferito per più tempo. Per il principio di località, dovrebbe essere anche la pagina a cui il processo non farà riferimento nei passi successivi; infatti, la strategia di LRU è quasi simile all'Algoritmo ottimo. Il problema con questo approccio è la difficoltà di implementazione: un approccio richiederebbe di contrassegnare ogni pagina con l'orario del suo ultimo riferimento; e questo dovrebbe essere fatto ad ogni riferimento di memoria, sia ad un'istruzione che a dei dati. Anche se l'hardware supportasse tale schema, sarebbe molto costoso; in alternativa, si potrebbe mantenere uno stack dei riferimenti alle pagine, ma sarebbe ancora costoso.

La Figura 8.13 mostra un esempio del comportamento di LRU usando la stessa sequenza di indirizzi a pagine usata nell'esempio dell'Algoritmo ottimo: in questi caso avvengono 4 fault di pagina. Un interessante raffinamento di LRU, SEQ, è stato proposto da [GIAS97]. Gli autori, esaminando un certo numero di applicazioni che fanno un intenso uso della memoria (cioè fanno molti riferimenti a memoria), hanno scoperto che mentre molte applicazioni mostrano una forte località, c'è anche un certo numero di applicazioni che mostrano chiare sequenze di riferimenti non locali, che possono essere sfruttate per migliori decisioni di sostituzione. Per questi ultimi programmi, intervalli di spazi di indirizzo sono percorsi ripetutamente in sequenza; generalmente sono applicazioni basate su array. Alcuni programmi percorrono ripetutamente intervalli della memoria in una sola direzione, mentre altri percorrono l'intervallo in entrambe le direzioni (avanti e indietro).

Un esame di un gran numero di tracce di programmi con queste sequenze di riferimenti sequenziali, mostra che LRU fornisce prestazioni scadenti. Se un programma accede ad un intervallo di indirizzi superiori a quelli della memoria allocata per il programma, LRU scarica le pagine nell'ordine di riferimento e perciò incontra un'infinita sequenza di fault di pagina. È utile osservare che anche aumentando la allocazione di memoria, non si riduce la percentuale di fault di pagina fino a che l'intero programma non ha trovato posto in memoria.

Gli autori idearono uno schema che è una variante del Most Recently Used (MRU) per integrare la strategia LRU. In sostanza, il sistema a paginazione cerca l'occorrenza di lunghe sequenze di riferimenti a pagine e usa le seguenti regole:

1. Una sequenza è una serie di fault di pagina ad indirizzi virtuali consecutivi, crescente in una direzione, senza fault a pagine intermedie. La pagina aggiunta più recentemente è la *testa* della sequenza.
2. Quando la memoria è poca e una pagina deve essere sostituita, il gestore della memoria determina se c'è una sequenza di lunghezza superiore a L. Esamina il tempo degli N fault più recenti in ogni sequenza e sceglie la sequenza la cui N-esima fault è la più recente.
3. Si sceglie per la sostituzione la prima pagina in memoria che sia M o più pagine distante dalla testa della sequenza.

L'algoritmo può essere messo a punto modificando L, N e M. Nelle applicazioni verificate, gli autori hanno trovato che i valori ($L = 20$, $N = 5$, $M = 20$) erano adeguati.

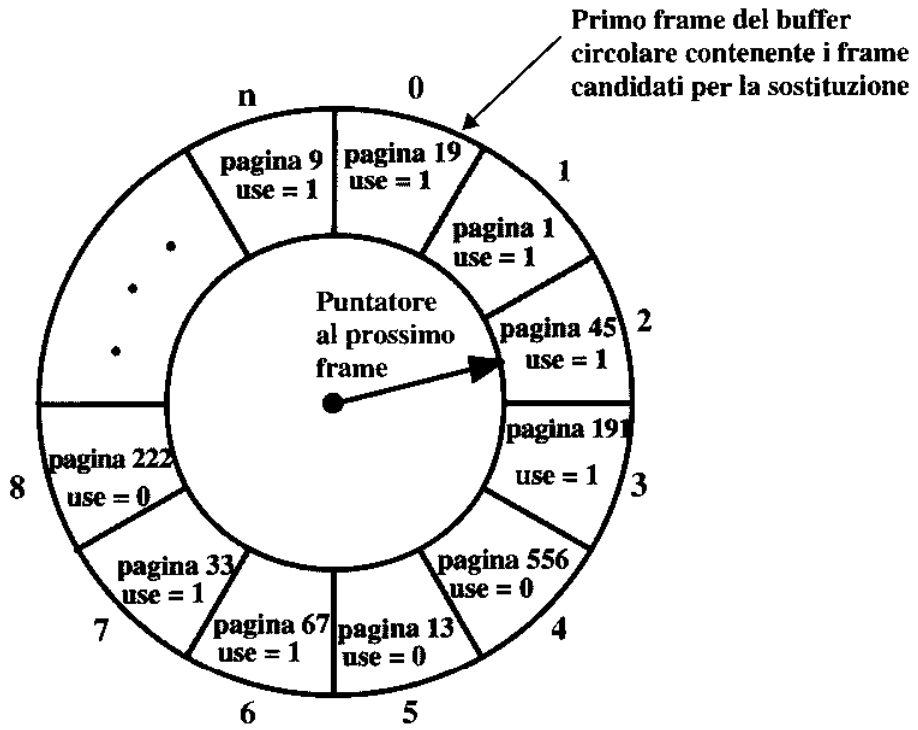
La strategia del **first-in first-out** tratta i frame allocati per le pagine di un processo come un buffer circolare e le pagine sono rimosse utilizzando lo schema del round-robin. Tutto ciò che è richiesto è un puntatore che va in cerchio lungo i frame contenenti le pagine del processo. Questa è perciò una delle strategie di sostituzione di pagine più semplici da implementare. La logica che sta dietro a questa scelta, oltre alla semplicità, è che si va a sostituire la pagina che è stata più a lungo in memoria: una pagina caricata in memoria da molto tempo può essere fuori uso. Questo ragionamento sarà spesso sbagliato perché ci saranno spesso porzioni di programma o dati molto usati durante tutta l'esecuzione di un programma; tali pagine saranno caricate e scaricate dall'algoritmo di FIFO

Continuando l'esempio di Fig.8.13, la strategia FIFO provoca 6 fault di pagina. Bisogna notare che LRU riconosce che le pagine 2 e 5 hanno riferimenti più frequenti delle altre pagine, mentre FIFO no.

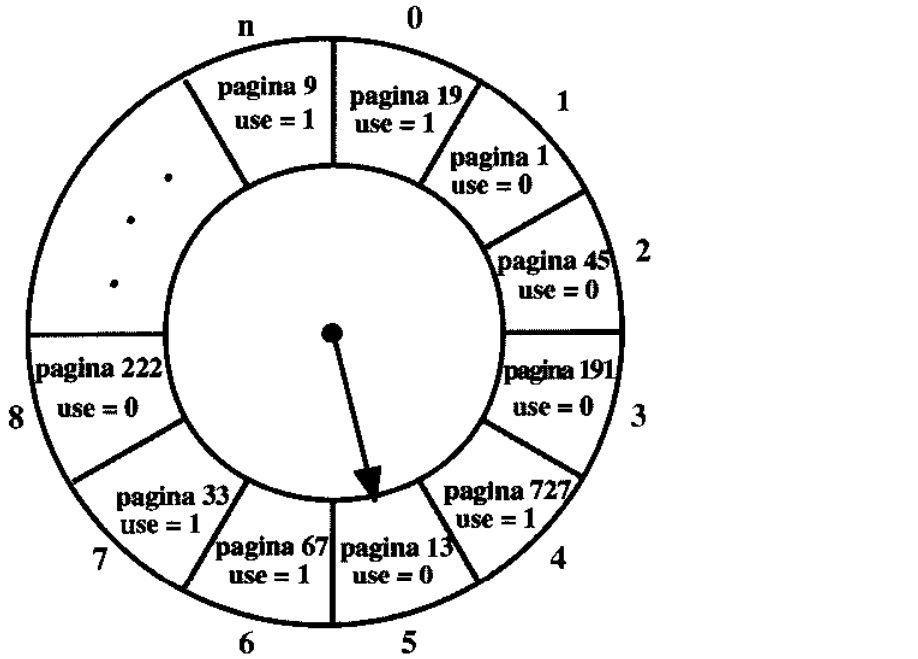
Anche se la strategia LRU fa bene quasi quanto la strategia ottimale, è difficile da implementare ed impone un overhead significativo. D'altra parte, la strategia FIFO è molto facile da implementare ma ha prestazioni relativamente scarse. Durante gli anni, i progettisti di sistemi operativi hanno provato diversi altri algoritmi per approssimare le prestazioni di LRU, imponendo un piccolo overhead. Molti di questi algoritmi sono varianti di uno schema che è la strategia dell'orologio.

La più semplice forma di strategia dell'orologio richiede l'associazione di un bit addizionale ad ogni frame, chiamato use bit. Quando una pagina è caricata per la prima volta in un frame di memoria, lo use bit di quel frame è fissato a 1. Quando ci si riferirà successivamente alla pagina (dopo il riferimento che ha generato un fault di pagina), il suo use bit sarà fissato a 1. Per l'algoritmo di sostituzione di pagine, l'insieme dei frame candidati alla sostituzione (quelli del processo corrente: ambito locale; tutta la memoria principale: ambito globale) è considerato come un buffer circolare, a cui è associato un puntatore. Quando una pagina è sostituita, il puntatore punta al frame successivo del buffer; quando viene il momento di sostituire una pagina, il sistema operativo scorre il buffer per trovare un frame il cui use bit è fissato a zero; se prima incontra un frame con use bit a 1, gli riassegna il valore zero. Se lo use bit di qualche frame del buffer ha valore zero all'inizio di questo ciclo, il primo di tali frame che si incontra è scelto per la sostituzione. Se tutti gli use bit dei frame hanno valore 1, allora il puntatore farà un giro completo del buffer, assegnando il valore zero a tutti gli use bit, e si fermerà nella sua posizione originale, sostituendo la pagina in quel frame. Si può notare che questa strategia è simile alla FIFO, eccetto il fatto che qualunque frame con use bit a 1 è evitato dall'algoritmo. La strategia è chiamata strategia dell'orologio perché è possibile visualizzare i frame di pagina come se fossero stesi su un cerchio. Un certo numero di sistemi operativi ha utilizzato alcune variazioni di questa semplice strategia dell'orologio (per esempio, Multics [CORB78]).

La Figura 8.14 mostra un esempio di funzionamento della semplice strategia dell'orologio. Un buffer circolare di n frame in memoria principale è disponibile per la sostituzione delle pagine. Proprio prima della sostituzione di una pagina del buffer con la pagina entrante 727, il puntatore alla pagina successiva punta al frame 2, che contiene pagina 45. Si esegue la strategia dell'orologio; poiché lo use bit di pagina 45 nel frame 2 è uguale a 1, questa pagina non è sostituita, allo use bit si assegna valore zero e il puntatore avanza. Similmente, non si sostituisce pagina 191 nel frame 3; al suo use bit si assegna valore zero e il puntatore avanza. Nel frame successivo, il frame 4, lo use bit ha valore zero; perciò pagina 556 è sostituita con pagina 727. Si



(a) Stato del buffer appena prima della sostituzione di pagina



(b) Stato del buffer subito dopo la sostituzione di pagina

Figura 8.14 *Esempio della strategia dell'orologio*

assegna allo use bit del frame il valore 1 e il puntatore raggiunge il frame 5, completando la procedura di sostituzione di pagine.

Il comportamento della strategia dell'orologio è illustrato in Figura 8.13. La presenza di un asterisco indica che il corrispondente use bit è uguale a 1, e la freccia indica la posizione corrente del puntatore. Bisogna notare che la strategia dell'orologio è adatta alla protezione dei frame 2 e 5 dalla sostituzione.

La Figura 8.15 mostra i risultati di un esperimento riportato in [BAER80], che confronta i quattro algoritmi discussi; si suppone che il numero dei frame assegnati al processo sia fissato. I risultati sono basati sull'esecuzione di 0.25 x 10^6 riferimenti in un programma FORTRAN, usando una dimensione di pagina di 256 parole. Baer ha eseguito l'esperimento allocando 6, 8, 10, 12 e 14 frame. Le differenze tra le quattro strategie sono più marcate utilizzando piccole allocazioni, con la strategia FIFO che è peggiore di oltre un fattore 2 dell'ottimale. Quasi identici risultati sono stati riportati in [FINK88], anche qui mostrando un'espansione massima di circa un fattore 2. L'approccio di Finkel simulava gli effetti delle diverse strategie su una stringa sintetizzata di 10000 riferimenti, selezionati da uno spazio virtuale di 100 pagine. Per approssimare gli effetti del principio di località, è stata imposta una distribuzione esponenziale di probabilità di riferirsi ad una particolare pagina. Finkel osserva che alcuni potrebbero concludere che non sembra molto utile implementare gli algoritmi di sostituzione di pagine quando è in gioco solo un fattore 2, ma nota che la differenza avrà un effetto visibile sia sui requisiti della memoria principale (per evitare peggioramenti nelle prestazioni di un sistema operativo) sia sulle prestazioni del sistema operativo (per evitare di espandere la memoria principale).

L'algoritmo dell'orologio è stato anche confrontato con questi altri algoritmi quando si utilizza una allocazione variabile e ambito sia locale che globale (vedere la successiva discussione sulle strategie di sostituzione) [CARR81, CARR84]. È stato riconosciuto che l'algoritmo dell'orologio approssima strettamente le prestazioni di LRU.

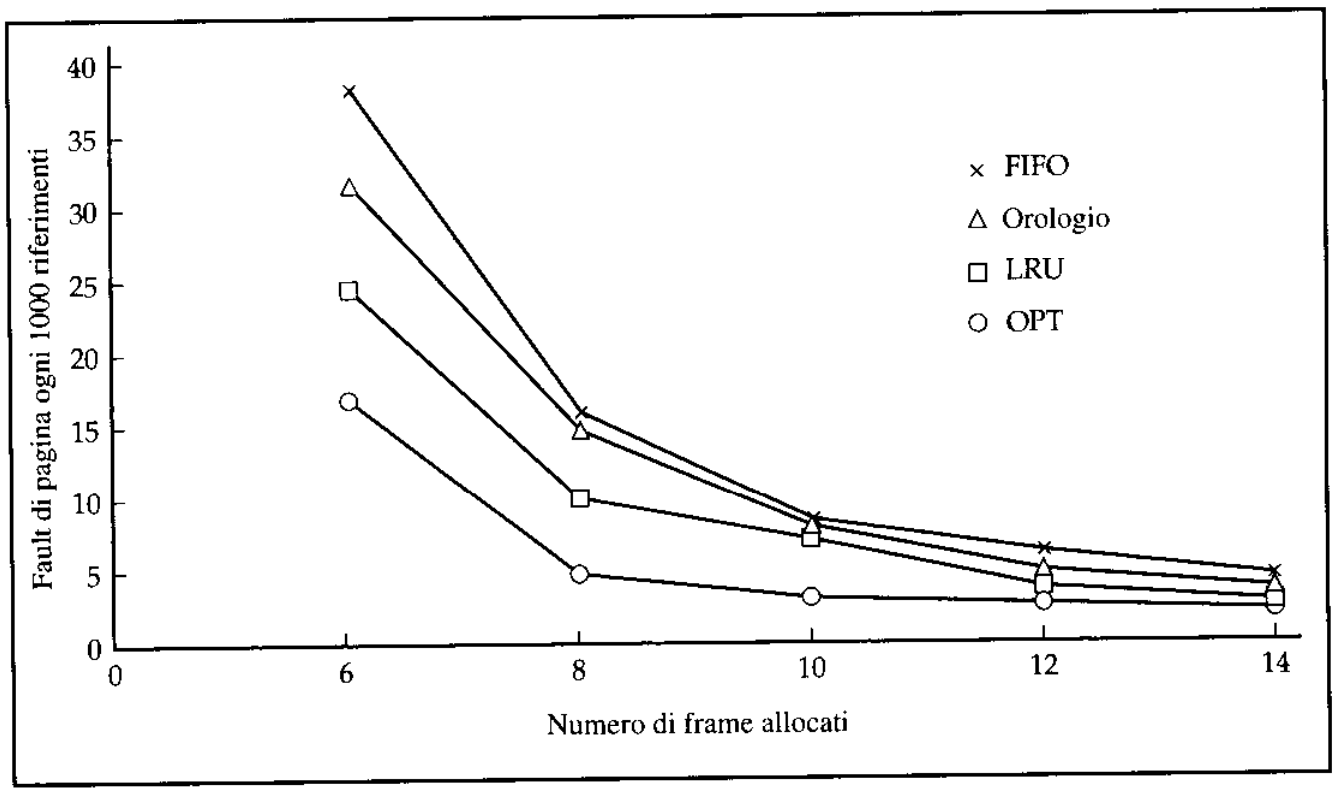


Figura 8.15 *Confronto degli algoritmi di sostituzione locale ad allocazione fissa*

L'algoritmo dell'orologio può essere reso più potente incrementando il numero di bit utilizzati[5]. In tutti i processori che supportano la paginazione, si associa un modify bit ad ogni pagina nella memoria principale e quindi ad ogni frame della memoria principale. Questo bit è necessario perché, quando si modifica una pagina, essa può essere sostituita solo dopo che è stata riscritta nella memoria secondaria. È possibile sfruttare questo bit nell'algoritmo dell'orologio nel seguente modo: se si tengono in considerazione lo use bit e il modify bit, ogni frame cade in una delle quattro seguenti categorie:

- Non utilizzato recentemente, non modificato ($u = 0$, $m = 0$)
- Utilizzato recentemente, non modificato ($u = 1$, $m = 0$)
- Non utilizzato recentemente, modificato ($u = 0$, $m = 1$)
- Utilizzato recentemente, modificato ($u = 1$, $m = 1$)

Con questa classificazione, l'algoritmo dell'orologio funziona nel seguente modo:

1. Cominciando dalla posizione a cui punta il puntatore, scorre il buffer. Durante lo scorrimento, non modifica il valore dello use bit. Non appena incontra un frame con ($u = 0$, $m = 0$) lo seleziona per la sostituzione.
2. Se il passo 1 fallisce, scorre nuovamente il buffer, cercando un frame con ($u = 0$, $m = 1$). Il primo frame che incontra con tale caratteristica è selezionato per la sostituzione. Durante lo scorrimento, assegna allo use bit di ogni frame il valore 0.
3. Se il passo 2 fallisce, il puntatore dovrebbe puntare nuovamente alla posizione di partenza e tutti gli use bit dei frame avranno valore 0. Ripete il passo 1 e, se necessario il passo 2: questa volta troverà un frame per la sostituzione.

Riassumendo, l'algoritmo di sostituzione di pagine cicla su tutte le pagine del buffer cercandone una che non sia stata ancora modificata da quando è stata caricata, e non sia stata usata recentemente. Tale pagina è un buona candidata per la sostituzione ed ha il vantaggio che, poiché non è stata ancora modificata, non ha bisogno di essere riscritta nella memoria secondaria. Se non si trova nessuna pagina candidata al primo passaggio, l'algoritmo è ripetuto ancora sul buffer cercando una pagina modificata a cui non si è acceduti recentemente. Anche se una pagina deve essere scaricata per essere sostituita, per il principio di località, può non essere subito di nuovo necessaria. Se questo seconda scansione fallisce, tutti i frame del buffer sono marcati come se non fossero stati usati recentemente, e si fa una terza scansione.

La strategia è utilizzata nello schema di memoria virtuale del Macintosh [GOLD89], illustrata nella Figura 8.16. Il vantaggio di questo algoritmo sull'algoritmo dell'orologio, sta nel fatto che le pagine che non sono state modificate sono preferite per la sostituzione.

Poiché una pagina che è stata modificata deve essere scritta su disco prima di essere sostituita, c'è un immediato risparmio di tempo.

---

[5] D'altra parte, se si riduce il numero di bit utilizzati a zero, l'algoritmo dell'orologio degenera in FIFO.

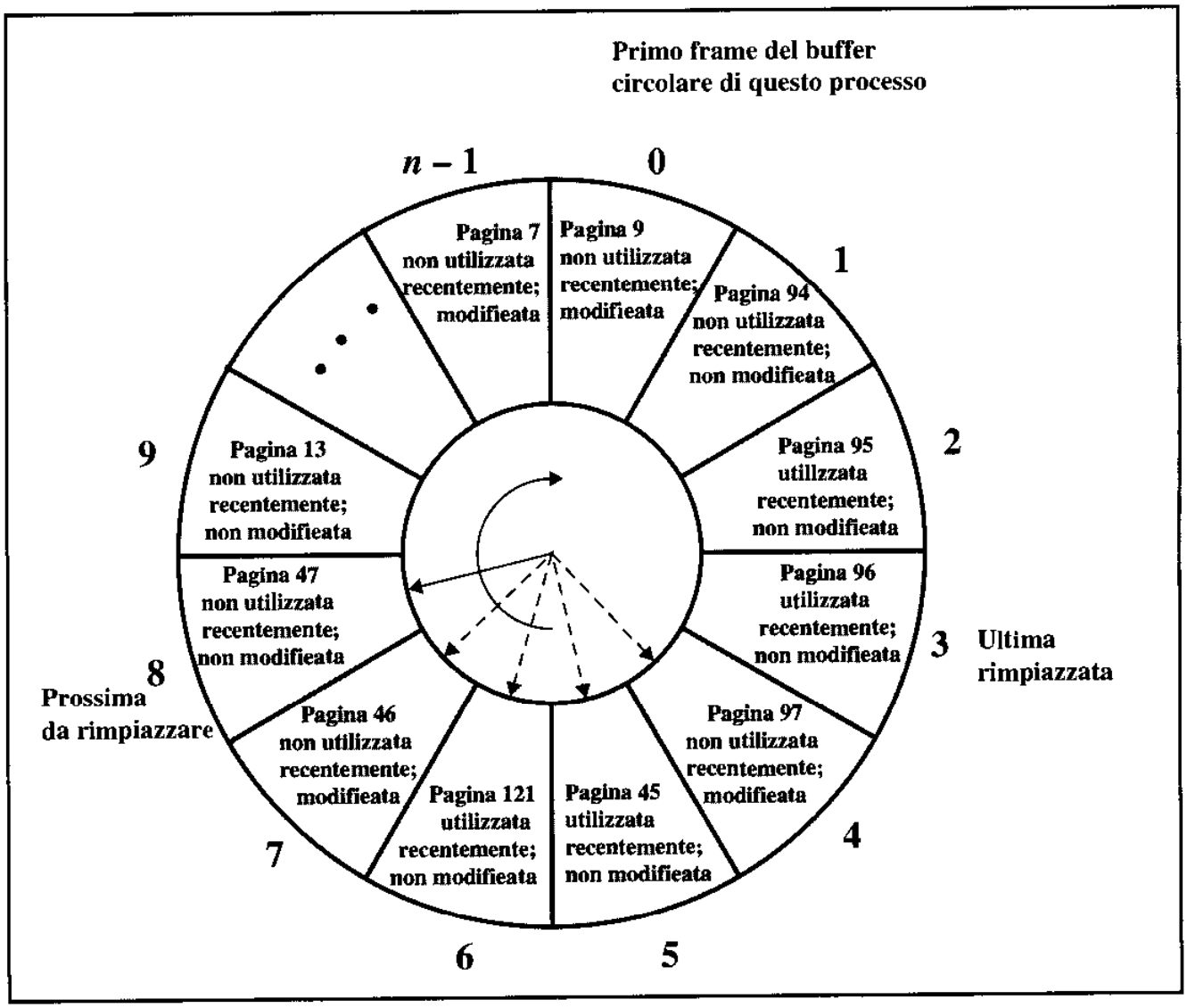


Figura 8.16 *Algoritmo dell'orologio per la sostituzione di pagine [GOLD89]*

Uso di buffer per pagine

Anche se le strategie LRU e dell'orologio sono superiori a FIFO, entrambe hanno problemi di complessità e overhead che FIFO non presenta. Inoltre, c'è il problema che il costo di sostituzione di una pagina che è stata modificata è maggiore di quello di una che non lo è stata, perché la prima deve essere riscritta nella memoria secondaria.

Un'interessante strategia che può migliorare le prestazioni della paginazione e permettere l'uso di una strategia di sostituzione di pagine più semplice è l'uso di buffer per pagine. L'approccio VAX VMS è il tipico rappresentante: VMS fa uso di allocazione variabile e della strategia di sostituzione locale. L'algoritmo di sostituzione di pagine è il semplice FIFO. Per migliorare le prestazioni, una pagina non più usata non è persa ma piuttosto è assegnata a una lista: alla lista delle pagine libere, se la pagina non è stata modificata, oppure alla lista delle pagine modificate. Bisogna notare che la pagina non è fisicamente spostata nella memoria principale; la sua entry nella tabella delle pagine è trasferita nella lista delle pagine libere o nella lista delle pagine modificate.

La lista delle pagine libere è una lista di frame disponibili per caricare le pagine. VMS cerca di tenere alcune pagine libere in ogni momento. Quando una pagina deve essere caricata, si usa il frame all'inizio della lista, sostituendo la pagina ivi residente; quando una pagina non modificata sta per essere sostituita, in realtà rimane in memoria e il suo frame è aggiunto in coda alla lista delle pagine libere. Similmente, quando una pagina modificata sta per essere sostituita, il suo frame è aggiunto in coda alla lista delle pagine modificate.

L'aspetto importante di queste manovre è che la pagina da sostituire rimane in memoria; perciò se il processo fa nuovamente riferimento a quella pagina, costa poco rimetterla nel resident set. In effetti, le liste delle pagine libere e modificate si comportano come una cache di pagine. La lista delle pagine modificate esegue un'altra utile funzione: le pagine modificate sono scritte su disco tutte insieme piuttosto che una alla volta questo riduce significativamente il numero di operazioni di I/O e perciò il tempo di accesso a disco.

Una versione più semplice del buffer per pagine è implementata nel sistema operativo Mach [RASH88]. In questo caso non esiste distinzione tra pagine modificate e pagine non modificate.

Strategia di sostituzione e dimensione della cache

Come è già stato osservato, la dimensione della memoria principale tende ad aumentare e la località delle applicazioni tende a diminuire; in compenso, le dimensioni della cache aumentano. Grandi dimensioni di cache, anche di più megabyte, sono oggi alternative di progettazione fattibili[BORG90]. Con una grande cache, la sostituzione delle pagine di memoria virtuale può avere un impatto sulle prestazioni: se il frame selezionato per la sostituzione è nella cache, allora quel blocco della cache è perso insieme con la pagina che contiene.

Nei sistemi che usano alcune forme di buffer per pagine, è possibile migliorare le prestazioni della cache integrando la strategia di sostituzione di pagine con una strategia per il posizionamento della pagina nel buffer. La maggior parte dei sistemi operativi posiziona le pagine selezionando un frame arbitrario dal buffer; solitamente si utilizza la tecnica del first-in, first-out, ma uno studio riportato in [KESS92] mostra che una accurata strategia di memorizzazione delle pagine può portare a diminuire del 10-20% il numero di accessi falliti alla cache, in confronto ad un metodo banale.

Diversi algoritmi di gestione della cache sono esaminati in [KESS92]. I dettagli sono oltre gli scopi di questo libro, poiché dipendono dai dettagli di struttura e strategie della cache. L'essenza di queste strategie sta nel caricare in memoria principale pagine consecutive, in modo da minimizzare il numero di frame che sono mappati nelle stesse entry di cache.

Gestione del resident set

Dimensione del resident set

Con la memoria virtuale gestita a pagine, non è necessario (e infatti può non essere possibile) caricare tutte le pagine di un processo nella memoria principale per prepararlo all'esecuzione; perciò il sistema operativo deve decidere quante pagine caricare - cioè quanta memoria principale allocare per un particolare processo. Entrano in gioco diversi fattori:

- Minore è la quantità di memoria allocata per un processo, maggiore è il numero di processi che può risiedere nella memoria principale in ogni momento. Questo aumenta la probabilità che il sistema operativo trovi almeno un processo Ready in ogni momento, e quindi riduce il tempo perso trasferendo processi su disco.
- Se un numero relativamente piccolo di pagine di un processo è nella memoria principale, allora, nonostante il principio di località, la frequenza di fault di pagina sarà piuttosto alta (Figura 8.9b).
- Oltre una certa dimensione, l'ulteriore allocazione di memoria principale per un particolare processo non avrà effetto visibile sulla frequenza di fault di pagina per quel processo, per il principio di località.

Con questi fattori in mente, i sistemi operativi contemporanei presentano due tipi di strategie. La strategia di **allocazione fissa** assegna al processo un numero fissato di pagine per l'esecuzione, deciso al momento del primo caricamento (tempo di creazione del processo) e può essere determinato in base al tipo di processo (interattivo, batch, tipo di applicazione) o in base alle indicazioni del programmatore o del sistemista. Con la strategia di allocazione fissa, ogni qualvolta avviene un fault di pagina nell'esecuzione del processo, una delle pagine di quel processo deve essere sostituita dalla pagina necessaria.

La strategia di **allocazione variabile** consente di variare il numero di frame allocati per un processo durante la sua esecuzione. Idealmente, un processo che dà sempre origine ad un elevato numero di fault di pagina, per il quale il principio di località sussiste in forma debole, potrà disporre di frame addizionali per ridurre il numero di fault di pagina, mentre, ad un processo con un numero eccezionalmente basso di fault di pagina, quindi buono dal punto di vista della località, sarà assegnata una minore allocazione di memoria, con la speranza che questo non porti ad un incremento visibile del numero di fault di pagina. L'uso della strategia di allocazione variabile è in relazione con l'ambito di sostituzione, come sarà spiegato nella successiva sottosezione.

La strategia di allocazione variabile sembra essere la più potente; comunque la difficoltà di questo approccio sta nel fatto che essa richiede che il sistema operativo giudichi il comportamento dei processi attivi, causando un inevitabile overhead software nel sistema operativo, ed è dipendente dai meccanismi hardware forniti dalla piattaforma del processore.

Ambito di sostituzione

L'ambito di una strategia di sostituzione può essere classificato come globale oppure come locale. Entrambi i tipi di strategia sono attivati da fault di pagina dovuti a mancanza di frame liberi. Una **strategia di sostituzione locale** sceglie la pagina da sostituire tra le pagine residenti in memoria del processo che ha generato fault di pagina. Una **strategia di sostituzione globale** considera tutte le pagine non bloccate in memoria principale come candidate alla sostituzione, senza badare a quale processo appartiene una particolare pagina. Mentre le strategie locali sono facili da analizzare, non è detto che abbiano prestazioni superiori a quelle delle strategie globali, che invece sono attraenti perché facili da implementare e con minimo overhead. [CARR84, MAEK87].

C'è una relazione tra l'ambito di sostituzione e la dimensione del resident set (Tabella 8.4).

Un resident set fissato implica una strategia di sostituzione locale: per mantenere fissa la dimensione del resident set, una pagina rimossa dalla memoria principale deve essere sostituita da un'altra pagina dello stesso processo. Una politica di allocazione variabile può chiaramente utilizzare una strategia di sostituzione globale: la sostituzione di una pagina di un processo nella memoria principale con una pagina di un altro processo fa si che la memoria allocata per un processo aumenti di una pagina e che la memoria allocata per l'altro processo si restringa di una pagina. Vedremo anche che è vantaggioso combinare l'allocazione variabile e la sostituzione locale. Esaminiamo queste tre combinazioni.

Allocazione fissa, ambito globale

In questo caso, abbiamo un processo in esecuzione nella memoria principale con un numero fisso di pagine a sua disposizione. Quando avviene un fault di pagina, il sistema operativo deve scegliere, tra le pagine del processo residenti in memoria al momento, la pagina da sostituire: si possono usare gli algoritmi di sostituzione appena discussi.

Con una politica di allocazione fissa, è necessario decidere in anticipo la quantità di allocazione da assegnare ad un processo, in base al tipo di applicazione e alla quantità di memoria richiesta dal programma. L'inconveniente di questo approccio è duplice: se l'allocazione tende ad essere troppo piccola, allora ci saranno numerosi fault di pagina, che rallentano il funzionamento dell'intero sistema in multiprogrammazione. Se invece le allocazioni tendono ad essere inutilmente grandi, ci saranno pochissimi processi nella memoria principale e quindi il processore lavorerà lentamente, oppure passerà un tempo considerevole a fare trasferimenti su disco dei processi.

Allocazione variabile, ambito globale

Questa combinazione è forse la più facile da implementare, ed è stata adottata da un certo numero di sistemi operativi. In ogni momento, ci sono dei processi nella memoria principale, ciascuno con a disposizione un certo numero di frame allocati. Generalmente il sistema operativo mantiene anche una lista dei frame liberi. Quando avviene un fault di pagina, un frame libero

Tabella 8.4 *Gestione del resident set*

	Sostituzione locale	**Sostituzione globale**
Allocazione fissa	Il numero di frame allocati per un processo è fisso, la pagina da rimpiazzare è scelta tra i frame allocati per il processo	Impossibile
Allocazione variabile	Il numero di frame allocati per un processo può essere cambiato nel tempo, per mantenere il working set del processo La pagina da sostituire è scelta tra i frame allocati per il processo	La pagina da sostituire è scelta tra tutti i frame disponibili in memoria principale; questo fa sì che la dimensione del resident set del processo possa variare

è aggiunto al resident set di un processo e si carica la pagina; perciò, un processo che genera fault di pagina crescerà gradualmente in dimensione, il che dovrebbe aiutare a ridurre il numero complessivo di fault di pagina del sistema.

La difficoltà con questo approccio sta nella scelta della sostituzione. Quando non ci sono frame liberi disponibili, il sistema operativo deve scegliere una pagina in memoria da sostituire tra tutti i frame in memoria, eccetto quelli bloccati, come quelli del kernel. Usando una delle strategie precedentemente illustrate, la pagina scelta per la sostituzione può appartenere a qualunque processo residente: non c'è un criterio per determinare quale processo perderà una pagina dal suo resident set. Perciò il processo che subirà una riduzione della dimensione del resident set può non essere il più adatto.

Un modo per opporsi ai potenziali problemi di prestazione della strategia di allocazione variabile e ambito globale è usare buffer per pagine. In questo modo la scelta della pagina da sostituire diventa meno significativa, perché la pagina può essere reclamata se è usata prima che il blocco di pagine di cui fa parte sia sovrascritto.

Allocazione variabile, ambito locale

La strategia di allocazione variabile, ambito locale tenta di superare i problemi incontrati con la strategia di ambito globale. Può essere sintetizzata nel seguente modo:

1. Quando un nuovo processo è caricato nella memoria principale, si alloca per lui un certo numero di frame, che dipende dal tipo di applicazione, dalle richieste del programma, o da altri criteri. Si usa la prepaginazione o la paginazione a richiesta per riempire i frame allocati
2. Quando avviene un fault di pagina, si seleziona la pagina da sostituire tra quelle del resident set del processo che provoca il fault di pagina.
3. Periodicamente, si rivaluta l'allocazione fornita al processo e la si aumenta o decrementa per migliorare le prestazioni globali.

Con questa strategia, la decisione di incrementare o decrementare la dimensione del resident set è deliberatamente basata su una valutazione delle probabili richieste future dei processi. A causa di questa valutazione, tale strategia è più complessa di una semplice strategia di sostituzione globale. Comunque, può dare migliori prestazioni.

Gli elementi chiave delle strategie di allocazione variabile e ambito globale sono il criterio usato per determinare la dimensione del resident set e la periodicità dei cambiamenti. Una strategia specifica che ha ricevuto molta attenzione in letteratura è conosciuta come **strategia del working set** (*working set strategy*). Sebbene una vera strategia del working set incontrerebbe difficoltà in fase di implementazione, è utile esaminarla come una base di confronto.

Il working set, introdotto e diffuso da Denning [DENN68, DENN70, DENN80b], ha avuto un profondo impatto sul modello per la gestione della memoria virtuale. Il working set con parametro Δ per un processo al tempo virtuale t, $W(t, \Delta)$, è l'insieme di pagine di quel processo a cui ci si è riferiti nelle ultime Δ unità di tempo virtuale. Il tempo virtuale è il tempo che passa mentre il processo è in esecuzione, ad esempio misurato in cicli di istruzione, cioè ogni istruzione eseguita è uguale ad un'unità di tempo.

Si consideri ciascuna delle due variabili di W. La variabile Δ è una finestra di tempo in cui si osserva il processo. La dimensione del working set sarà una funzione non decrescente della dimensione della finestra. Il risultato è mostrato in Figura 8.17 (basato su [BACH86]), che mostra una sequenza di riferimenti a pagina per un processo. I puntini indicano le unità di tempo in cui il working set non cambia. Bisogna notare che più grande è la dimensione della finestra, più grande è il working set. Questo può essere espresso dalla seguente relazione:

$$W(t, \Delta + 1) \supseteq W(t,\Delta)$$

Il working set è anche una funzione del tempo. Se un processo è eseguito oltre le Δ unità di tempo e usa solo una pagina, allora $|W(t, \Delta)| = 1$. Un working set può anche crescere fino ad essere grande quanto il numero delle N pagine del processo se si accede a molte pagine diverse rapidamente e se la dimensione della finestra lo permette. Perciò

$$1 \leq | W(t,\Delta | \leq \min(\Delta,N)$$

La Figura 8.18 indica il modo in cui la dimensione del working set può variare nel tempo per un valore fissato di Δ. Per molti programmi, periodi con dimensioni di working set relativamen-

Sequenza di riferimenti a pagina	Dimensione della finestra, Δ			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	18 24	17 24 18	24 17 18	15 24 17 18

Figura 8.17 *Working set di un processo definito dalla dimensione della finestra*

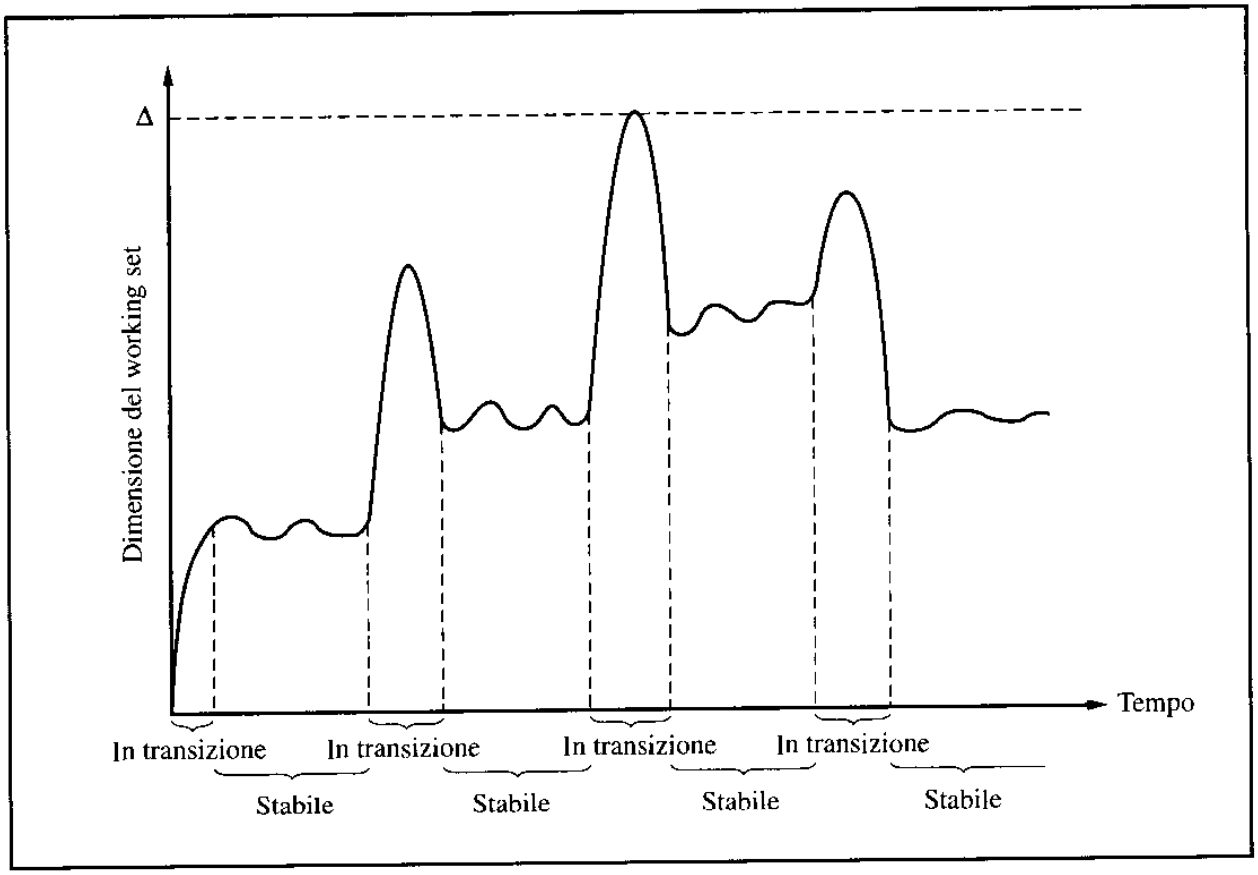


Figura 8.18 *Grafico tipico della dimensione di un vero working set [MAEK87]*

te stabili si alternano a periodi con rapidi cambiamenti. Quando un processo comincia la sua esecuzione, costruisce gradatamente il suo working set mentre fa riferimento a nuove pagine. Prima o poi, per il principio di località, il processo dovrebbe stabilizzarsi su un certo insieme di pagine; successivi periodi di transizione riflettono uno spostamento del processo verso una nuova località. Durante la fase di transizione, alcune pagine dalla vecchia località rimangono all'interno della finestra Δ, causando la crescita della dimensione del working set quando si fa riferimento ad una nuova pagina. Quando poi la finestra supera questi riferimenti a pagina, la dimensione del working set diminuisce fino a contenere solo le pagine della nuova località.

Questo concetto di working set può essere usato per guidare una strategia per la dimensione del resident set:

1. Controllare il working set di ogni processo.
2. Rimuovere periodicamente dal resident set di un processo quelle pagine che non sono nel suo working set.
3. Un processo può essere eseguito solo se il suo working set è nella memoria principale (cioè se il suo resident set contiene il suo working set).

Questa strategia è interessante perché prende un concetto accettato, il principio di località, e lo sfrutta per ottenere una strategia di gestione della memoria che minimizza il numero di fault

di pagina. Sfortunatamente, ci sono dei problemi correlati alla strategia del working set:

1. Il passato non predice sempre il futuro. Sia la dimensione del working set, sia l'appartenenza dei frame al working set cambiano nel tempo (ad esempio vedere la Figura 8.18).
2. Non è pratico stimare la dimensione del working set per ogni processo. Sarebbe necessario marcare con un time stamp ogni riferimento a pagina per ogni processo, usando il tempo virtuale di quel processo, e poi mantenendo una coda di pagine per ogni processo ordinata in base al tempo.
3. Il valore ottimale di Δ è sconosciuto e in ogni caso è soggetto a variazioni.

Tuttavia, lo spirito di questa strategia è valido, e un certo numero di sistemi operativi tentano di approssimare la strategia del working set. Un modo per farlo è concentrarsi non sull'esatto numero di riferimenti a pagina, ma sulla frequenza di fault di pagina di un processo. Come mostra la Figura 8.9b, la frequenza di fault di pagina scende incrementando la dimensione del resident set di un processo. La dimensione del working set diminuirebbe in un punto della curva indicato con W nella Figura. Perciò, piuttosto che controllare la dimensione del working set direttamente, si possono ottenere risultati confrontabili controllando la frequenza di fault di pagina. La linea di ragionamento è la seguente: se la frequenza di fault di pagina è al di sotto di una certa soglia, l'intero sistema può migliorare assegnando al processo un resident set più piccolo (perché più frame sono disponibili per altri processi) senza danneggiare il processo (causando più fault di pagina). Se la frequenza di fault di pagina è più della soglia superiore, il processo può beneficiare di un aumento della dimensione del resident set (provocando meno fault di pagina) senza degradare il sistema.

Un algoritmo che segue questa strategia è l'**algoritmo di page-fault frequency** (PFF) (frequenza dei fault di pagina) [CHU72, GUPT78]. L'algoritmo richiede un use bit da associare ad ogni pagina presente in memoria: quando si accede alla pagina, si assegna il valore 1 allo use bit. Quando avviene un fault di pagina, il sistema operativo prende nota del tempo virtuale trascorso dall'ultimo fault di pagina per il processo considerato: questo si può fare mantenendo un contatore dei riferimenti alla pagina. Si definisce una soglia F: se il tempo trascorso dall'ultimo fault di pagina è minore di F, allora si aggiunge una pagina al resident set del processo, altrimenti, si scartano tutte la pagine che hanno lo use bit con valore zero e si restringe di conseguenza il resident set. Allo stesso tempo, si riassegna il valore zero agli use bit delle pagine rimanenti. La strategia può essere raffinata usando due soglie: una soglia superiore che è usata per provocare un aumento della dimensione del resident set, e una soglia inferiore per contrarre la dimensione del resident set.

Il tempo tra i fault di pagina è il reciproco della frequenza di fault di pagina. Sebbene sembrerebbe meglio aggiornare la media della frequenza di fault di pagina, farne una sola misura è un ragionevole compromesso che permette di decidere la dimensione del resident set in base alla frequenza di fault di pagina. Se tale strategia è integrata con l'uso di buffer per pagine, le prestazioni sono abbastanza buone.

Tuttavia, c'è un difetto nell'approccio del PFF: non ha buone prestazioni durante la fase di transizione quando c'è uno spostamento ad una nuova località. Con il PFF, nessuna pagina esce dal resident set prima che siano trascorse F unità di tempo virtuale dall'ultima volta in cui sono

state oggetto di riferimento. Durante le transizioni tra località, la rapida successione dei fault di pagina provoca un aumento del resident set di un processo prima che le pagine della vecchia località siano espulse; gli improvvisi picchi di richiesta di memoria possono produrre inutili disattivazioni e riattivazioni dei processi, con i corrispondenti overhead indesiderati.

Un approccio che tenta di arginare questo fenomeno della transizione di località con un overhead relativamente basso, simile a quello del PFF, è la **strategia di variable-interval sampled working set(VSWS)** (working set campionato ad intervalli variabili) [FERR83]. La strategia di VSWS valuta il working set di un processo ad intervalli di campionamento del tempo virtuale trascorso. All'inizio dell'intervallo di campionamento, gli use bit di tutte le pagine del processo residenti in memoria sono azzerati; alla fine, solo le pagine a cui il processo ha fatto riferimento durante l'intervallo contengono nello use bit il valore 1, e queste pagine saranno mantenute nel resident set nel successivo intervallo, mentre le altre saranno scartate. Perciò la dimensione del resident set può solo decrescere alla fine dell'intervallo. Durante ogni intervallo, tutte le pagine che hanno causato fault di pagina sono aggiunte al resident set; perciò il resident set rimane fissato o cresce durante l'intervallo.

La strategia di VSWS è guidata dai tre parametri:

M: durata minima dell'intervallo di campionamento

L: durata massima dell'intervallo di campionamento

Q: numero di fault di pagina consentiti tra gli intervalli di campionamento

La strategia VSWS procede nel modo seguente:

1. Se il tempo virtuale dall'ultimo campionamento raggiunge L, allora sospende il processo e controlla gli use bit.
2. Se, prima che sia trascorso un tempo virtuale L, avvengono Q fault di pagina,
   a. Se il tempo virtuale dall'ultimo campionamento è minore di M, allora aspetta fino a che il tempo virtuale non raggiunge M, per sospendere il processo e controllare gli use bit.
   b. Se il tempo virtuale dall'ultimo campionamento è maggiore o uguale a M, sospende il processo e controlla gli use bit.

I valori dei parametri devono essere selezionati in modo che il campionamento sia normalmente scatenato dall'occorrenza dell'i-esimo fault di pagina successivo all'ultimo controllo degli use bit (caso 2b). Gli altri due parametri (M e L) forniscono limiti di protezione da condizioni particolari. La strategia di VSWS cerca di ridurre i picchi di richieste di memoria causati da inaspettate transizioni di località, aumentando la frequenza di campionamento, e quindi la frequenza a cui le pagine inutilizzate escono dal resident set, quando la frequenza di paginazione aumenta. L'esperienza fatta con questa tecnica dal sistema operativo per i mainframe Bull, GCOS8, indica che questo approccio è semplice da implementare quanto il PFF, ma più efficace [PIZZ89].

Strategia di cleaning

La strategia di cleaning è l'opposto della strategia di fetch; determina quando una pagina modificata deve essere scritta nella memoria secondaria. Due comuni alternative sono cleaning a richiesta e precleaning. Con cleaning a richiesta, una pagina è scritta nella memoria secondaria, solo quando è stata scelta per essere sostituita. La strategia di precleaning scrive le pagine modificate prima che i loro frame siano necessari, così che le pagine possono essere scritte a gruppi.

C'è un pericolo nel seguire una delle due strategie fino all'estremo. Con il precleaning, una pagina è scritta nella memoria secondaria, ma rimane nella memoria principale fino a che l'algoritmo di sostituzione di pagine non ordina che sia rimossa. Il precleaning consente la scrittura delle pagine in gruppi, ma ha poco senso scrivere su disco centinaia o migliaia di pagine, solo per scoprire che la maggioranza di esse è stata ancora modificata prima di essere sostituita. La capacità di trasferimento della memoria secondaria è limitata e non dovrebbe essere sprecata con inutili operazioni di cleaning.

D'altro canto, con il cleaning a richiesta, la scrittura di una pagina modificata precede la lettura di una nuova pagina. Questa tecnica può minimizzare il numero delle scritture di pagina, ma significa che un processo che ha fault di pagina può dover aspettare due trasferimenti di pagine per poter essere sbloccato, diminuendo l'uso del processore.

Un approccio migliore utilizza buffer per pagine, con la seguente strategia: cleaning solo per le pagine che sono sostituibili, dissociando le operazioni di cleaning e sostituzione. Con l'uso di buffer per pagine, le pagine sostituite possono essere inserite in due liste: la lista delle pagine modificate e la lista delle pagine non modificate. Le pagine della prima lista possono essere periodicamente scritte in batch e spostate nella seconda lista. Una pagina non modificata può essere richiamata se il processo fa riferimento ad essa, o persa quando il suo frame è assegnato ad un'altra pagina.

Controllo del carico

Il controllo del carico determina il numero dei processi che sarà residente nella memoria principale, a livello di multiprogrammazione. La strategia del controllo del carico è critica per una gestione efficiente della memoria: se in memoria sono residenti pochissimi processi in un certo momento, ci saranno molte occasioni in cui tutti i processi sono bloccati, e si passa molto tempo a trasferirli su disco. D'altro canto, se ci sono troppi processi residenti, allora, in media, la dimensione del resident set di ogni processo sarà inadeguata e avverranno frequenti fault: il risultato è il thrashing.

Livello di multiprogrammazione

Il thrashing è illustrato in Figura 8.19. Facendo crescere il livello di multiprogrammazione, ci si aspetterebbe di veder crescere l'uso del processore, perché ci sono meno probabilità che tutti i processi siano bloccati. In seguito, si raggiunge un punto in cui il resident set medio risulta inadeguato: la frequenza di fault di pagina cresce drasticamente e l'uso del processore collassa.

Ci sono diversi modi per superare questo problema. Un algoritmo del working set, o di

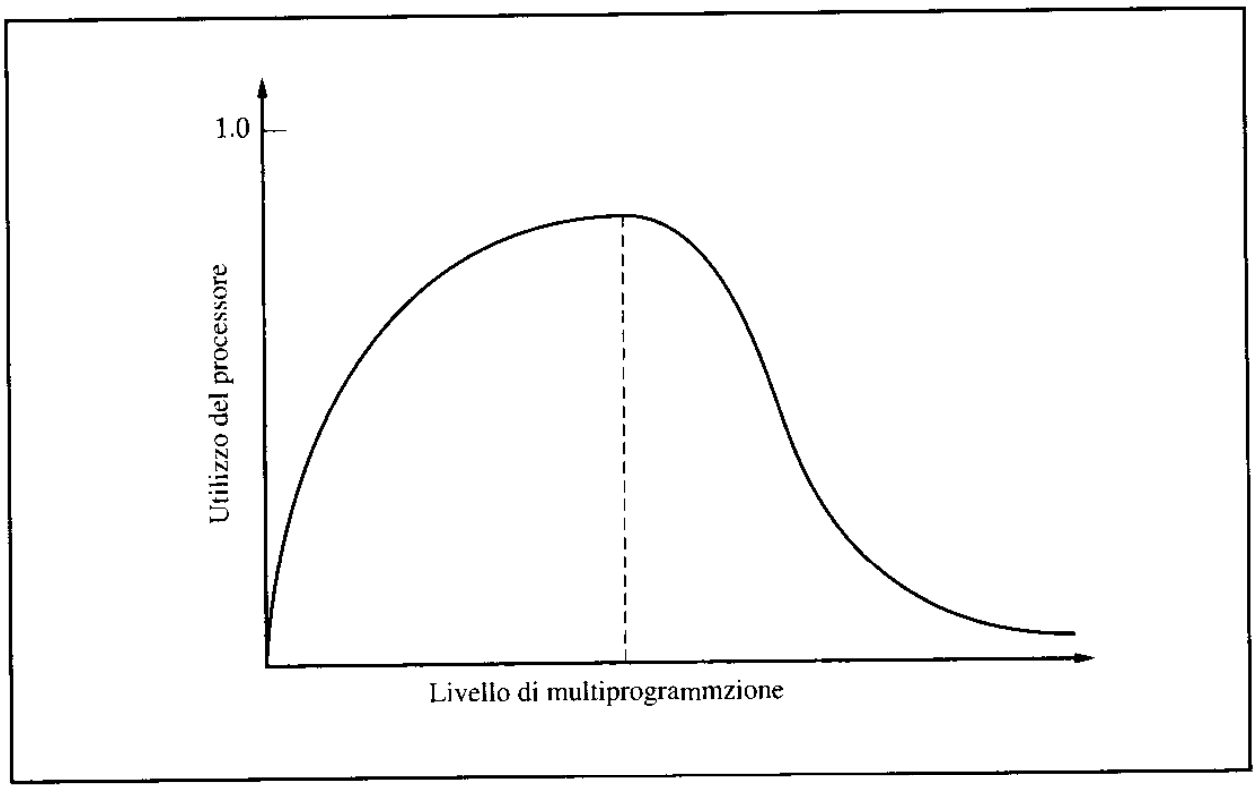


Figura 8.19 *Effetti della multiprogrammazione*

frequenza di fault di pagina, incorpora implicitamente il controllo del carico, eseguendo solo quei processi i cui resident set sono sufficientemente grandi. Fornendo un resident set della dimensione richiesta per ogni processo attivo, la strategia determina automaticamente e dinamicamente il numero dei processi attivi.

Un altro approccio, suggerito da Denning e i suoi colleghi [DENN80b], conosciuto come "criterio $L = S$", regola il livello di multiprogrammazione in modo tale che il tempo medio tra i fault sia uguale al tempo medio richiesto per gestire una fault di pagina. Gli studi sulle prestazioni indicano che questo è il punto in cui l'uso del processore raggiunge il massimo. Una strategia con un simile effetto, proposta in [LEROY76], è il "criterio del 50%" che cerca di mantenere l'uso del dispositivo di paginazione approssimativamente intorno al 50%, infatti, gli studi sulle prestazioni indicano che questo è un punto di massimo uso del processore.

Un altro approccio richiede di adattare l'algoritmo dell'orologio per la sostituzione delle pagine descritto precedentemente (Figura 8.16). [CARR84] descrive una tecnica, usando un ambito globale, che controlla la frequenza con cui il puntatore scorre il buffer circolare dei frame. Se l'intervallo è al di sotto di una data soglia inferiore, questo indica una o entrambe le seguenti circostanze:

1. Stanno avvenendo pochi fault di pagina, infatti vi sono poche richieste di avanzamento del puntatore.
2. Per ogni richiesta, il numero medio di frame scorsi dal puntatore è piccolo, quindi ci sono molte pagine residenti a cui il processo non ha fatto riferimento e che quindi sono prontamente sostituibili.

In entrambi i casi, il livello di multiprogrammazione può essere aumentato con sicurezza. D'altra parte, se l'intervallo di scansione del puntatore supera una soglia superiore, questo indica che c'è una grande frequenza di fault, oppure una difficoltà a reperire le pagine sostituibili, il che implica che il livello di multiprogrammazione è troppo alto.

Sospensione del processo

Se il livello di multiprogrammazione deve essere ridotto, uno o più tra i processi residenti deve essere sospeso (scaricato su disco). [CARR84] elenca sei possibilità:

- **Processo con la priorità più bassa**: Questo implementa una decisione di scheduling e non ha correlazione con le prestazioni.
- **Processo che provoca fault**: Il ragionamento è che c'è una alta probabilità che il task che provoca il fault non abbia il suo working set residente in memoria, quindi, le prestazioni non ne risentono troppo se il processo viene sospeso. Inoltre, questa scelta ha un guadagno immediato, perché blocca un processo che è comunque sul punto di bloccarsi, ed elimina così l'overhead di sostituzione di pagine e delle operazioni di I/O.
- **Ultimo processo attivato**: Questo è il processo che ha meno probabilità di aver ancora il suo working set in memoria.
- **Processo con il più piccolo resident set**: Questo richiederà in futuro meno lavoro per essere ricaricato; comunque, penalizza i programmi con piccola località.
- **Processo più grande:** Ottiene il maggior numero di frame liberi in una memoria sovraccarica, rendendo meno probabili disattivazioni ulteriori per un po' di tempo.
- **Processo con la più grande finestra di esecuzione rimanente**: Nella maggior parte degli schemi di scheduling, un processo può essere eseguito solo per un certo quanto di tempo, prima di essere interrotto e spostato alla fine della coda dei processi Ready. Questo approssima una disciplina di shortest-processing-time-first scheduling.

Come in altre aree di progettazione dei sistemi operativi, la scelta della strategia da usare dipende da molti altri fattori della progettazione e dalle caratteristiche del processo che deve essere eseguito.

8.3 Gestione della memoria di UNIX e Solaris

Poiché il sistema UNIX è pensato per essere indipendente dalla macchina, il suo schema di gestione della memoria varierà da un sistema all'altro. Le prime versioni di UNIX usavano semplicemente il partizionamento variabile senza schema di memoria virtuale. Le implementazioni odierne, inclusi SVR4 e Solaris 2.x, fanno uso della memoria virtuale paginata.

In SVR4 e Solaris, ci sono attualmente due schemi separati per la gestione della memoria. Il **sistema di paginazione** (*paging system*) realizza una memoria virtuale, che alloca ai processi i

frame delle memoria principale e alloca anche i buffer del disco alle pagine di memoria virtuale. Sebbene questo sia uno schema di gestione delle memoria efficace per i processi utente e l'I/O su disco, uno schema di memoria virtuale paginata è poco adatto per gestire l'allocazione di memoria per il kernel. Per quest'ultimo scopo, si usa un **allocatore di memoria per il kernel**. Si veda ora un esame dei due meccanismi.

Sistema di paginazione

Strutture dati

Per la memoria virtuale paginata, UNIX fa uso di un certo numero di strutture dati che, con un minimo adattamento, è indipendente dalla macchina. (Figura 8.20 e Tabella 8.5):

- **Tabella delle pagine:** generalmente è presente una tabella delle pagine per processo, con una entry per ogni pagina del processo presente nella memoria virtuale.

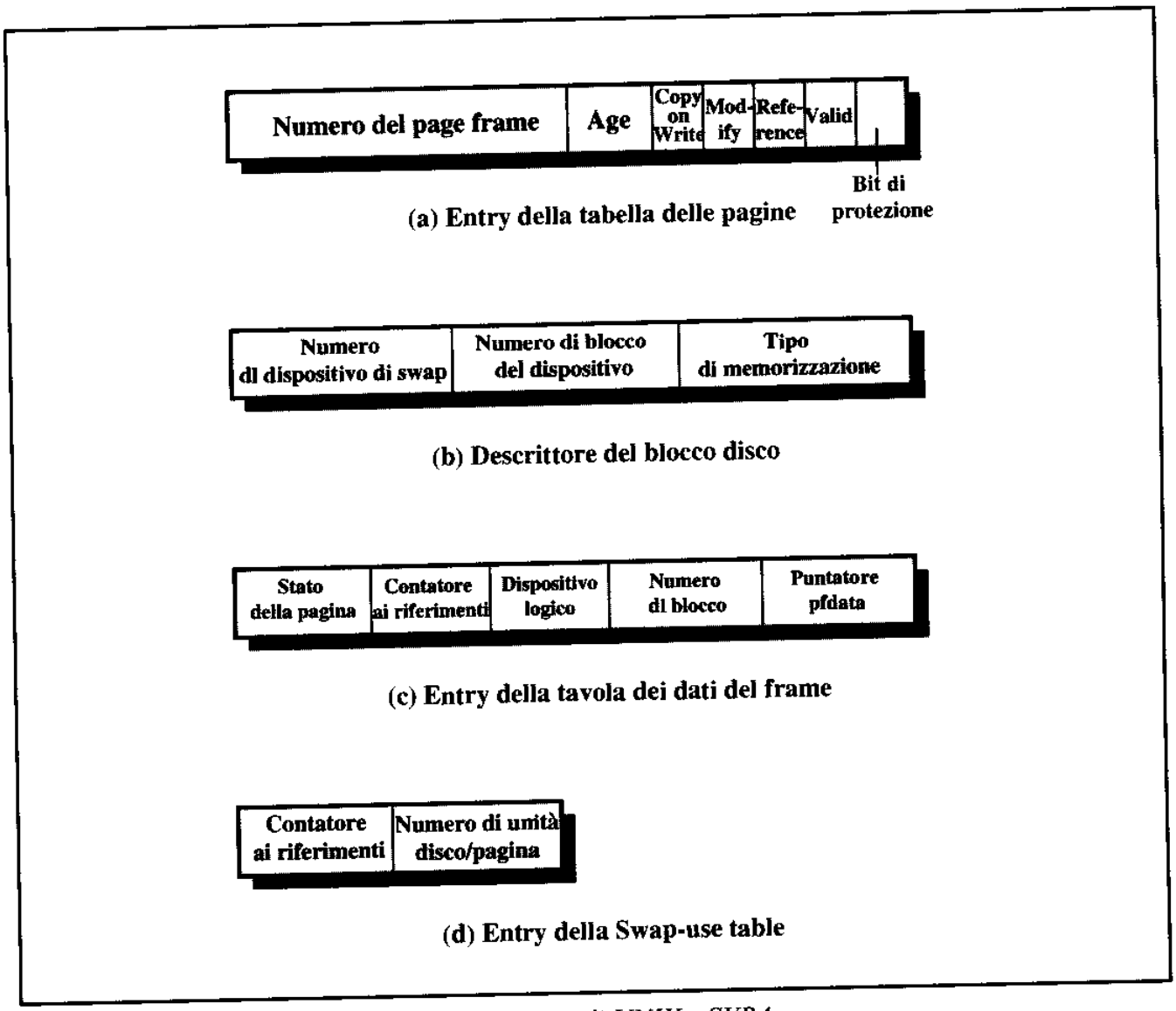


Figura 8.20 *Formati della gestione di memoria di UNIX e SVR4*

Tabella 8.5 *Parametri per la gestione della memoria di UNIX SVR4*

Entry della Page Table

Numero di page frame
Si riferisce ai frame della memoria reale

Age
Indica per quanto tempo la pagina è rimasta in memoria senza che il processo facesse riferimento ad essa. La lunghezza e i contenuti di questo campo dipendono dal processore.

Copy on write
È posto a 1 quando più di un processo condivide una pagina. Se uno dei processi scrive nella pagina, deve essere fatta una copia della pagina per tutti gli altri processi che la condividono. Questo consente di rimandare l'operazione di copiatura finché è necessaria, ed evitarla quando non è necessaria.

Modify
Indica se la pagina è stata modificata.

Reference
Indica se ci sono stati riferimenti alla pagina. Questo bit è posto a zero quando la pagina è caricata per la prima volta, e può essere periodicamente reimpostato a 0 dall'algoritmo di page replacement.

Valid
Indica la presenza della pagina nella memoria principale.

Protect
Indica se l'operazione di scrittura è consentita.

Descrittore del blocco disco

Numero del dispositivo di swap
Numero di dispositivo logico del dispositivo di memoria secondaria che contiene la pagina corrispondente. Questo consente di utilizzare più di un dispositivo per memorizzare pagine trasferite su disco.

Numero del blocco del dispositivo
Locazione del blocco che contiene la pagina sul dispositivo

Tipo di memorizzazione
La memorizzazione può essere un'unità di swap o un file eseguibile. Nell'ultimo caso, c'è un'indicazione sulla necessità di pulire la memoria virtuale prima di allocarla.

Entry della Frame Data Table

Stato della pagina
Indica se questo frame è disponibile o ha una pagina ad esso associata. In quest'ultimo caso, lo stato della pagina è specificato: sul device di swap, nel file eseguibile o DMA in corso.

Contatore ai riferimenti
Numero di processi che fanno riferimento alla pagina.

Dispositivo Logico
Dispositivo logico che contiene una copia della pagina

Numero di blocco
Locazione del blocco che contiene la copia della pagina sul dispositivo logico.

Puntatore a Pfdata
Puntatore agli altri entry della pfdata table su una lista di pagine libere o su una coda hash di pagine.

Entry della Swap-use Table

Contatore ai riferimenti
Numero degli entry della page table che puntano ad una pagina nel dispositivo di swap.

Numero di unità disco/pagina
Identificatore di pagina sull'unità di memoria.

- **Disk block descriptor:** Ad ogni pagina del processo è associata una entry di questa tabella che descrive la copia su disco della pagina virtuale.
- **Frame data table:** Descrive ogni frame della memoria reale e ha come indici i numeri dei frame.
- **Swap-use table:** C'è una swap-use table per ogni dispositivo su cui trasferire i processi, con una entry per ogni pagina del dispositivo.

La maggior parte dei campi definiti nella Tabella 8.5 si spiegano da soli, ma sono necessari alcuni ulteriori commenti. Il campo "Age" nella entry della tabella delle pagine indica quanto tempo è trascorso senza che un programma facesse riferimento al frame. Comunque, il numero di bit e la frequenza di aggiornamento di questo campo sono dipendenti dall'implementazione, non essendoci un uso universale in UNIX di questo campo per la strategia di sostituzione di pagine.

Il campo "Tipo di memorizzazione" nel descrittore del blocco disco è necessario per il seguente motivo: quando un file eseguibile è usato per la prima volta per creare un nuovo processo, solo una parte del programma e dei dati è caricata. In un secondo tempo, quando avviene un fault di pagina, nuove parti del programma e dei dati sono caricate. È solo al momento del primo caricamento che sono create le pagine di memoria virtuale e sono assegnate a locazioni su uno dei dispositivi da usare per il trasferimento dei processi. In quel momento, si informa il sistema operativo se deve pulire (porre a 0) le locazioni nel frame prima del primo caricamento di un blocco contenente il programma o i dati.

Sostituzione di pagine

La frame data table è usata per la sostituzione di pagine. Diversi puntatori sono usati per creare liste all'interno di questa tabella. Tutti i frame disponibili sono inseriti in una lista di frame liberi per il caricamento di pagine; quando il numero di frame disponibili scende al di sotto di una certa soglia, il kernel ruberà un numero di pagine per compensare.

L'algoritmo di sostituzione di pagine usato in SVR4 è un raffinamento dell'algoritmo della strategia dell'orologio (Figura 8.14) conosciuto come algoritmo dell'orologio con due lancette (Figura 8.21). L'algoritmo usa un reference bit nella entry della tabella delle pagine per ogni pagina in memoria idonea (non bloccata) per essere scaricata. Questo bit è posto a 0 quando la pagina è caricata in memoria per la prima volta, ed a 1 quando la pagina è riferita. Una lancetta nell'algoritmo dell'orologio, quella *anteriore*, scorre attraverso le pagine contenute nella lista delle pagine idonee e assegna il valore 0 al reference bit di ogni pagina. Successivamente, quella *posteriore* scorre la stessa lista e controlla il reference bit. Se il bit ha valore 1, allora la pagina ha avuto un riferimento dopo il passaggio della lancetta anteriore e il frame è ignorato. Se il bit ha ancora valore 0, allora il processo non ha più fatto riferimento alla pagina nel tempo trascorso tra la visita delle due lancette; queste pagine sono inserite in una lista per essere scaricate.

Due parametri determinano l'operazione dell'algoritmo:

- **Scanrate:** frequenza con cui le due lancette scorrono la lista delle pagine, in pagine per secondo
- **Handspread:** distanza tra lancetta anteriore e posteriore

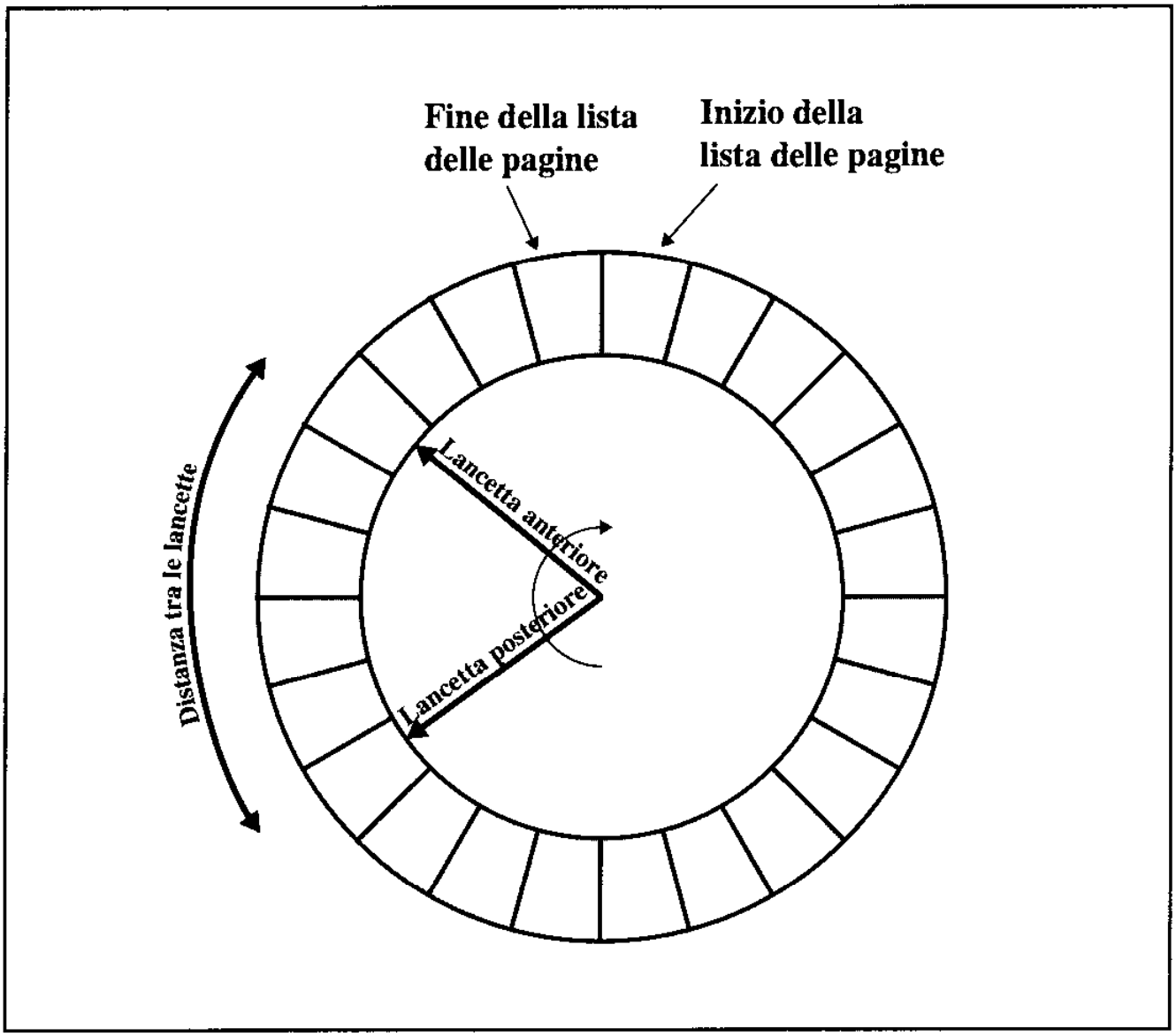


Figura 8.21 *Algoritmo dell'orologio con due lancette*

Questi due parametri hanno valori per difetto assegnati a tempo di boot, in base alla quantità di memoria fisica. Il parametro *scanrate* può essere alterato per adattarsi a condizioni variabili. Il parametro varia in modo lineare tra i valori *slowscan* e *fastscan* (assegnati al momento della configurazione) come la quantità di memoria libera varia tra i valori *lotsfree* e *minfree*. In altre parole, non appena aumenta la quantità di memoria libera, la lancetta si sposta più rapidamente per liberare più pagine. Il parametro handspread determina la distanza tra le due lancette, e perciò insieme a scanrate determina l'opportunità di usare una pagina prima che sia scaricata perché non usata.

Allocatore di memoria per il kernel

Il kernel genera e distrugge frequentemente piccole tabelle e buffer durante l'esecuzione, ognuna delle quali richiede un'allocazione dinamica della memoria. [VAHA96] elenca i seguenti esempi:

- L'instradamento della traduzione dei pathname può allocare un buffer per copiare un nome di path dallo spazio utente.
- La routine allocb() alloca STREAMS buffer di dimensione arbitraria.
- Molte implementazioni di UNIX allocano strutture zombie per conservare lo stato di uscita e informazioni sull'utilizzo delle risorse dei processi morti.
- In SVR4 e in Solaris, il kernel alloca molti oggetti (come proc structure, vnode, blocchi di descrittori di file) dinamicamente quando necessario.

La maggior parte di questi blocchi è significativamente più piccola della tipica dimensione di pagina della macchina, e perciò il meccanismo di paginazione sarebbe inefficiente per l'allocazione dinamica della memoria del kernel. Per SVR4 si usa una modifica del buddy system, descritto nella sezione 7.2.

Nei buddy system, il costo per allocare e liberare un blocco di memoria è basso se confrontato con quelli delle strategie di first-fit o best-fit [KNUT97]. Comunque nel caso della gestione della memoria del kernel, le operazioni di allocazione e liberazione devono essere fatte il più in fretta possibile. Il difetto del buddy system è il tempo richiesto per frammentare e riunire i blocchi.

Alla AT&T, Barkley e Lee hanno proposto una variazione conosciuta come buddy system pigro[BARK89], e questa è la tecnica adottata per SVR4. Gli autori hanno osservato che UNIX spesso presenta un comportamento stabile nella richiesta di memoria del kernel: cioè, la quantità di domanda per blocchi di una particolare dimensione varia lentamente nel tempo. Perciò, se un blocco di dimensione 2^i è rilasciato ed è subito riunito con il suo compagno in un blocco di dimensione 2^{i+1}, il kernel può successivamente richiedere un blocco di dimensione 2^i che fa dividere nuovamente il blocco più grande. Per evitare queste inutili riunioni e divisioni di blocchi, il buddy system pigro rimanda la riunione dei blocchi fino a che non è necessaria, e poi riunisce più blocchi possibili.

Il buddy system pigro usa i seguenti parametri:

N_i = numero di blocchi di dimensione 2^i

A_i = numero di blocchi di dimensione 2^i che sono allocati (occupati).

G_i = numero di blocchi di dimensione 2^i globalmente liberi: questi blocchi sono idonei per la riunione; se il compagno di tale blocco diventa globalmente libero, allora i due blocchi saranno riuniti per formare un unico blocco globalmente libero di dimensione 2^{i+1}. Tutti i blocchi liberi (holes) nel sistema buddy standard possono essere considerati globalmente liberi.

L_i = numero di blocchi con dimensione 2^i che sono localmente liberi: questi sono blocchi non idonei ad essere riuniti. Anche se il compagno di tale blocco diventa libero, i due blocchi non si riuniscono. Piuttosto, i blocchi localmente liberi sono mantenuti in previsione di future richieste di un blocco di tali dimensioni.

Vale la seguente relazione:

$$N_i = A_i + G_i + L_i$$

In generale il buddy system pigro cerca di mantenere un insieme di blocchi localmente liberi e invoca la riunione se il numero di blocchi localmente liberi supera una soglia stabilita. Se ci sono troppi blocchi localmente liberi, allora c'è la possibilità che manchino i blocchi liberi al livello superiore, per soddisfare le richieste. La maggior parte delle volte in cui un blocco è liberato, non avviene la riunione, così ci sono costi minimi di operazioni ed aggiornamenti. Quando un blocco deve essere allocato, non si fa distinzione tra blocchi liberi globalmente e localmente; ancora, questo minimizza gli aggiornamenti.

Il criterio usato per la riunione è che il numero di blocchi localmente liberi di una data dimensione non superi il numero di blocchi allocati di quella dimensione (ad esempio, è possibile avere $L_i \leq A_i$). Questo criterio limita ragionevolmente la crescita dei blocchi localmente liberi, e gli esperimenti in [BARK89] confermano che questo schema offre notevoli risparmi.

Per implementare lo schema, gli autori definiscono la seguente variabile di ritardo:

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

La Figura 8.22 mostra l'algoritmo.

Il valore iniziale di D_i è 0
Dopo un'operazione, il valore di D_i è aggiornato nel seguente modo:

(I) if la prossima operazione è una richiesta di allocazione di un blocco:
  if ci sono bloccchi liberi, se ne seleziona uno da allocare
    if il blocco allocato è localmente libero
     then $D_i := D_i + 2$
     else $D_i := D_i + 1$
  otherwise
    si ottengono due blocchi dividendo a metà uno blocco più grande (operazione ricorsiva)
    se ne alloca uno e si marcano gli altri come "localmente liberi"
    non si modifica D_i (ma D può cambiare per altre dimensioni di blocchi a causa della chiamata ricorsiva)

(II) if la prossima operazione è una richiesta di blocco libero
  Case $D_i \geq 2$
    lo si marca come "libero localmente" e lo si libera localmente
    $D_i := D_i - 2$
  Case $D_i = 1$
    lo si marca come "libero globalmente"; si riunisce se posssibile
    $D_i := 0$
  Case $D_i = 0$
    lo si marca come "libero globalmente" e lo si libera globalmente; si riunisce se possibile
    si seleziona un blocco localmente libero di dimensione 2i e lo si libera globalmente;
    si riunisce se possibile
    $D_i := 0$

Figura 8.22 *Algoritmo del pigro Buddy System*

8.4 La gestione della memoria in Windows NT

Il gestore della memoria per Windows NT controlla come è allocata la memoria e come è realizzata la paginazione. Il gestore della memoria è progettato per operare su diverse piattaforme e usa dimensioni di pagina che variano da 4KB a 64KB. Le piattaforme Intel, PowerPC e MIPs hanno 4096 byte per pagina e la piattaforma DEC Alpha ha 8192 byte per pagina.

Mappa degli indirizzi virtuali in NT

Ogni processo utente di NT vede uno spazio di indirizzi a 32 bit separato, dando fino a 4GB di memoria per processo. Per difetto, metà di questa memoria è riservata per il sistema operativo, così ogni utente ha a disposizione uno spazio di indirizzi virtuali di 2GB e tutti i processi condividono gli stessi 2GB dello spazio di sistema. NT 4.0 ha un'opzione che consente di estendere lo spazio utente a 3GB, lasciando un gigabyte per lo spazio di sistema. La documentazione che riguarda NT indica che questa caratteristica è stata aggiunta per supportare grandi applicazioni che occupano molto la memoria, su server con molti gigabyte di RAM, e che l'uso di spazi di indirizzo più grandi può accrescere notevolmente le prestazioni di applicazioni come supporto di decisione e data mining.

La Figura 8.23 mostra lo spazio di indirizzi virtuali disponibile per difetto, visto da un processo utente. Si compone di 4 regioni:

- Da 0x00000000 a 0x00000FFF: lasciati vuoti, per aiutare i programmatori a catturare le assegnazioni a NULL dei puntatori.
- Da 0x00010000 a 0x7FFEFFFF: disponibili per lo spazio di indirizzi utente. Questo spazio è diviso in pagine che possono essere caricate nella memoria principale.
- Da 0x7FFF0000 a 0x7FFFFFFF: Una pagina guardia inaccessibile dall'utente. Questa pagina semplifica il compito di controllo del sistema operativo sui puntatori a indirizzi fuori-limite.
- Da 0x80000000 a 0xFFFFFFFF: Spazio di indirizzi di sistema. Questo processo da 2GB è usato per NT Executive, il microkernel e i driver dei dispositivi.

La paginazione in NT

Quando un processo è creato, può, in principio, usare l'intero spazio utente da 2 gigabytea 128 kilobyte. Questo spazio è diviso in pagine di dimensione fissata, ciascuna delle quali può essere portata nella memoria principale. In pratica, per semplificare la contabilità, la pagina può essere in uno dei tre stati:

- **Disponibile:** Pagine non usate correntemente da questo processo.
- **Riservata:** Un insieme di pagine contigue che il gestore di memoria virtuale mette da parte per un processo, ma non mette in conto nella quota di memoria del processo fino a che non è usata. Quando un processo ha bisogno di scrivere in memoria, parte della memoria riservata è assegnata al processo.

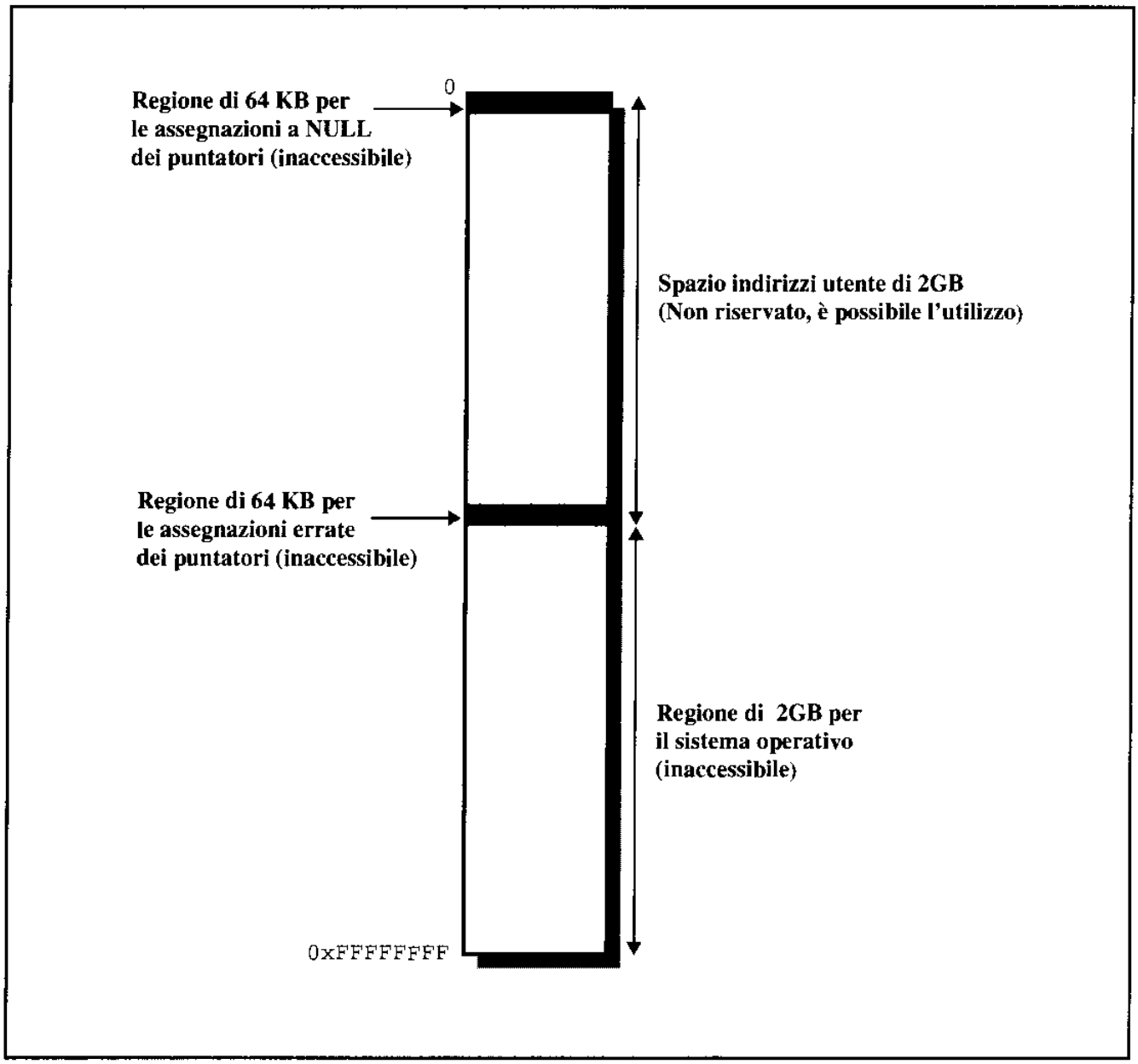


Figura 8.23 *Spazio di indirizzamento virtuale per difetto in Windows NT*

- **Committed:** Pagine per le quali il gestore di memoria virtuale ha riservato spazio nel suo file di paginazione, il file su disco sul quale scrive le pagine quando sono rimosse dalla memoria principale.

La distinzione tra la memoria riservata e committed è utile perché: (1) minimizza la quantità di spazio disco riservato per un particolare processo, lasciando lo spazio disco libero per altri processi, e (2) abilita un thread o processo a dichiarare una quantità di memoria che può essere velocemente allocata non appena è necessario.

Lo schema di gestione del resident set usato da Windows NT è allocazione variabile, ambito locale (vedere Tabella 8.4). Quando un processo è attivato per la prima volta, gli vengono assegnati un certo numero di frame della memoria principale come suo working set; quando un processo si riferisce ad una pagina non presente in memoria, una delle pagine residenti di quel

processo è scaricata e la nuova pagina è caricata. I working set dei processi attivi sono adattati usando le seguenti convenzioni generali:

- Quando c'è abbondanza di memoria principale, il gestore di memoria virtuale consente ai resident set dei processi attivi di crescere. Per fare questo, quando avviene un fault di pagina, una nuova pagina è caricata in memoria, ma nessuna vecchia pagina è scaricata, così si incrementa il working set di quel processo di una pagina.
- Quando la memoria comincia a scarseggiare, il gestore della memoria virtuale recupera memoria per il sistema togliendo dal working set dei processi attivi le pagine meno usate recentemente, riducendo la dimensione di quei resident set.

8.5 Sommario

Per usare efficientemente il processore e le facilitazioni di I/O, è desiderabile mantenere più processi possibili in memoria. Inoltre, si vuole liberare i programmatori dalle restrizioni di dimensione nello sviluppo di programmi.

Il modo di risolvere entrambi questi problemi è la memoria virtuale. Con la memoria virtuale, tutti i riferimenti di indirizzi sono riferimenti logici, che sono tradotti a tempo di esecuzione in indirizzi reali. Questo fa sì che un processo possa essere allocato in qualunque parte della memoria principale, e che la sua posizione possa cambiare nel tempo. La memoria virtuale, inoltre, permette ad un processo di essere spezzato. Questi pezzi non devono essere necessariamente contigui nella memoria principale durante l'esecuzione e, infatti, non è neanche necessario che tutti i pezzi del processo siano nella memoria principale durante l'esecuzione.

Due sono gli approcci principali alla memoria virtuale: la paginazione e la segmentazione. Con la paginazione, ogni processo è suddiviso in pagine relativamente piccole di dimensione fissata; la segmentazione consente l'uso di pezzi di dimensione variabile. È anche possibile combinare la paginazione e la segmentazione in un solo schema per la gestione della memoria.

Uno schema di gestione della memoria virtuale richiede sia supporti hardware sia software. Il supporto hardware è fornito dal processore: comprende la traduzione dinamica degli indirizzi virtuali in indirizzi fisici, e la generazione di un interrupt quando ci si riferisce ad una pagina o ad un segmento non presenti nella memoria principale. Tale interrupt attiva il software per la gestione della memoria nel sistema operativo.

Vi sono diversi problemi di progettazione in relazione al supporto che il sistema operativo offre per la gestione della memoria:

- **Strategia di fetch:** le pagine del processo possono essere caricate su richiesta, o può essere utilizzata una politica di prepaginazione, che raggruppa le attività di input caricando un certo numero di pagine alla volta.
- **Strategia di posizionamento:** con un sistema di pura segmentazione, un segmento che si vuole caricare deve essere posto in uno spazio di memoria disponibile.

- **Strategia di sostituzione:** quando la memoria è piena, si deve decidere quali e quante pagine sostituire.
- **Gestione del resident set:** il sistema operativo deve decidere quanta memoria allocare per un particolare processo che si carica. Si può fare una allocazione statica, al tempo di creazione del processo, o la si può cambiare dinamicamente.
- **Cleaning:** le pagine dei processi modificati possono essere scritte su disco al momento della sostituzione, o può essere usata una strategia di precleaning, che raggruppa l'attività di output scrivendo su disco un certo numero di pagine alla volta
- **Controllo del carico:** si basa sulla determinazione del numero di processi residenti nella memoria principale in ogni momento.

8.6 Letture raccomandate

Come ci si può aspettare, la memoria virtuale è largamente trattata nella maggior parte dei testi sui sistemi operativi. [MAEK87] riassume esaurientemente le varie aree di ricerca. [CARR84] esamina approfonditamente le tematiche di prestazione. [KUCK78] e [BAER80] presentano interessanti risultati analitici e di simulazione. Il testo classico [DENN70] è ancora degno di essere letto.

È una esperienza interessante leggere [IBM86], che fa un dettagliato resoconto degli strumenti e delle opzioni disponibili per il sistemista, che vuole ottimizzare le strategie di memoria virtuale di MVS. Il documento illustra la complessità del problema.

BAER80, Baer, J. *Computer Systems Architecture*. Rockville, MD: Computer Science Press, 1980.

CARR84, Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.

DENN70, Denning, P. "Virtual Memory" *Computing Surveys*, Settembre 1970.

IBM86, IBM National Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook*. Dallas Systems Center Technical Bulletin G320-0597, June 1986.

KUCK78, Kuck, D. *The Structure of Computers and Computations*. New York: Wiley, 1978.

MAEK87, Maekawa, M.; Oldehoeft, A.; e Oldehoeft, R. *Operating Systems: Advanced Concepts*. Menlo Park, CA: Benjamin Cummings, 1988.

8.7 Problemi

8.1 Si supponga che la tabella delle pagine di un processo in esecuzione sul processore sia del tipo illustrato in figura. Tutti i numeri sono decimali, tutto è numerato a partire da zero, e tutti gli indirizzi sono di byte in memoria. La dimensione della pagina è di 1024 byte.

Numero della pagina virtuale	Valid bit	Reference bit	Modify bit	Numero di frame
0	1	1	0	4
1	1	1	1	7
2	0	0	0	-
3	1	0	0	2
4	0	0	0	-
5	1	0	1	0

a. Descrivere esattamente come, in generale, un indirizzo virtuale generato dalla CPU è tradotto in un indirizzo fisico della memoria principale.

b. A quale indirizzo fisico, se esiste, corrisponde ciascuno dei seguenti indirizzi virtuali? (Non cercare di trattare il fault di pagina, nel caso capitasse).

(i)1052 (ii) 2221 (iii) 5499

8.2 Un processo dispone di quattro frame. (Tutti i numeri sono decimali, ogni cosa è numerata a partire da zero.) Nella seguente tabella sono indicati: il tempo dell'ultimo caricamento di una pagina in ogni frame, il numero di pagina virtuale nel frame, il tempo dell'ultimo accesso alla pagina in ogni frame, il reference bit (R) e il modify bit (M) per ogni frame (il tempo è indicato in colpi di clock dall'inizio del processo, al momento 0, fino all'evento – non è il numero di colpi dal momento dall'evento al momento presente).

Numero della pagina virtuale	Page Frame	Tempo di caricamento	Tempo di riferimento	R bit	M bit
2	0	60	161	0	1
1	1	130	160	0	0
0	2	26	162	1	0
3	3	20	163	1	1

È avvenuto un fault di pagina per la pagina virtuale 4. Quale frame è sostituito secondo ciascuna delle seguenti strategie di gestione della memoria? Spiegare il perché in ogni caso.

a. FIFO (first-in-first-out)

b. LRU (least recently used)

c. NRU (not used recently). (Bisogna usare i bit M e R in modo ragionevole – e bisogna tenere presente che, se si vuole capire come è sorta questa particolare configurazione, tutti i reference bit sono stati posti a zero per l'ultima volta dopo il colpo di clock 161.)

d. Orologio

e. L'algoritmo ottimo di Belady (usare la seguente stringa di riferimenti.)

f. Dato lo stato della memoria prima del fault di pagina, si consideri la seguente stringa di riferimenti alla pagina virtuale.

4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2

Quanti fault di pagina avverrebbero se la strategia del working set fosse usata con una finestra di dimensione 4 invece che con un'allocazione con dimensione fissata? Mostrare chiaramente quando avverrebbe ogni fault di pagina.

8.3 Un processo fa riferimento a cinque pagine, A, B, C, D ed E nell'ordine seguente

A; B; C; D; A; B; E; A; B; C; D; E

Si supponga di utilizzare l'algoritmo di sostituzione first-in-first-out e si trovi il numero di trasferimenti di pagina che devono essere eseguiti durante questa sequenza di riferimenti, partendo con una memoria principale vuota contenente tre frame. Ripetere l'esercizio per il caso di memoria con quattro frame.

8.4 Nel VAX, le tabelle delle pagine utente sono allocate ad indirizzi virtuali nello spazio di sistema. In cosa sta il vantaggio di avere la tabelle delle pagine utente nella memoria virtuale anziché in quella principale?

8.5 Si supponga di eseguire l'istruzione

```
for i in 1....n do A[i] : = B[i] + C[i] endfor
```

in una memoria in cui la dimensione delle pagine è pari a 1000 parole. Sia $n = 1000$. Usando una macchina che ha a disposizione un insieme completo di istruzioni registro-registro e che utilizza registri indice, scrivere un ipotetico programma per implementare l'istruzione precedente. Mostrare la sequenza dei riferimenti a pagina che avvengono durante l'esecuzione.

8.6 L'architettura IBM System/370 usa una struttura di memoria a due livelli, chiamati rispettivamente segmenti e pagine, anche se ciò che si intende per segmento non ha tutte le caratteristiche discusse in questo capitolo. Per l'architettura base 370, la dimensione della pagina può essere o di 2KB o di 4KB, e la dimensione del segmento è fissata a 64KB o a 1MB. Quali vantaggi della segmentazione non sono presenti in questo schema? Qual è il vantaggio della segmentazione per il 370?

8.7 Si consideri una pagina con dimensione di 4KB e che ogni entry della tabella delle pagine occupi 4 byte. Quanti livelli di tabella delle pagine sarebbero richiesti per mappare uno spazio di indirizzo di 64 bit, se il livello più alto della tabella delle pagine sta in una sola pagina?

8.8 Si consideri un sistema che gestisce la memoria per pagine e che usa una tabella delle pagine a singolo livello. Si supponga che la tabella delle pagine necessaria sia sempre in memoria.

a. Se un riferimento a memoria è eseguito in 200 ns, quanto costerà, in termini di tempo, fare riferimento ad una memoria paginata?

b. Si aggiunga un MMU che impone un overhead di 20 ns per usare il TLB, sia per successo sia per fallimento. Se si suppone che l'85% di tutti i riferimenti a memoria abbiano successo nel TLB, qual è il tempo effettivo di accesso a memoria?

c. Spiegare come la frequenza di successo nel TLB influisce sul tempo effettivo di accesso alla memoria.

8.9 Si consideri una stringa di riferimenti a pagina per un processo con un working set di M frame, inizialmente tutti vuoti. La stringa di riferimenti a pagina è di lunghezza P con N numeri di pagina diversi. Per ogni algoritmo di sostituzione di pagine

a. Qual è il limite inferiore sul numero di fault di pagina?

b. Qual è il limite superiore sul numero di fault di pagina?

8.10 Nell'architettura S/370, una chiave di memorizzazione è un campo di controllo associato ad ogni frame della memoria reale. Due bit di quella chiave che sono significativi per la sostituzione di pagine sono il reference bit e il change bit. Il reference bit è posto a 1 quando si accede a un qualunque indirizzo del frame per lettura o scrittura, ed è posto a 0 quando si carica una nuova pagina nel frame. Il change bit è posto a 1 quando si esegue un'operazione di scrittura su una qualunque locazione all'interno del frame. Suggerire un approccio per determinare quali frame sono stati usati meno recentemente, facendo uso del solo reference bit.

8.11 Un punto chiave per le prestazioni della strategia di gestione del resident set di VSWS è il valore di Q. L'esperienza ha mostrato che, con un valore fissato di Q per un processo, ci sono considerevoli differenze nella frequenza di fault di pagina nei diversi stadi di esecuzione. Inoltre, se lo stesso valore di Q è usato per diversi processi, le frequenze dei fault di pagina saranno drasticamente diverse. Queste differenze indicano che un meccanismo che variasse decisamente il valore di Q durante l'esecuzione di un processo migliorerebbe il comportamento dell'algoritmo. Suggerire un semplice meccanismo per questo scopo.

8.12 Si supponga che un task sia diviso in 4 segmenti di uguale misura e che il sistema costruisca una page descriptor table di 8 campi per ogni segmento. Perciò il sistema sfrutta una combinazione di segmentazione e paginazione. Si supponga che la pagina abbia una dimensione di 2KB.

a. Qual è la dimensiona massima di ogni segmento?

b. Qual è il massimo spazio di indirizzi logici per il task?

c. Si supponga che il task acceda ad un elemento nella locazione di memoria 00021ABC. Qual è il formato dell'indirizzo logico che il task gli crea?

d. Qual è il massimo spazio di indirizzi fisici per il sistema?

8.13 Si consideri uno spazio di indirizzi "paginato" (composto da 32 pagine da 2KB ciascuna), mappato in un spazio di memoria fisica da 1MB.

a. Qual è il formato dell'indirizzo logico del processore?

b. Quali sono la lunghezza e la larghezza della tabella delle pagine (ignorando i bit dei "diritti di accesso")?

c. Qual è l'effetto di un dimezzamento dello spazio di memoria fisica sulla tabella delle pagine?

8.14 Un computer ha una cache, una memoria principale e un disco usato per la memoria virtuale. Se una parola cui si fa riferimento è nella cache, ci vogliono 20ns per avere accesso ad essa. Se è nella memoria principale ma non nella cache, sono necessari 60 ns per caricarla nella cache, e poi riparte il riferimento. Se la parola non è nella memoria principale, sono richiesti 12 millisecondi per recuperare la parola dal disco, seguiti da 60 ns per copiarla sulla cache, e poi riparte il riferimento. La frequenza di successo della cache è 0.9 e la frequenza di successo della memoria principale è 0.6. Qual è il tempo medio in ns necessario per accedere ad una parola riferita in questo sistema?

Appendice 8A Le tavole hash

Si consideri il seguente problema: un insieme di N elementi deve essere memorizzato in una tabella. Ogni elemento è composto da un'etichetta più alcune informazioni aggiuntive, che chiameremo il valore dell'oggetto. Si vorrebbe essere in grado di eseguire un certo numero di operazioni standard sulla tabella, come l'inserimento, la cancellazione e la ricerca di un elemento in base alla sua etichetta.

Se gli elementi sono etichettati con numeri compresi nell'intervallo da 0 a M-1, allora una semplice soluzione consiste nell'utilizzare una tabella di lunghezza M. L'elemento etichettato con i dovrà essere inserito nella locazione i della tabella. Poiché gli elementi sono di lunghezza fissata, la ricerca nella tabella è banale e comporta l'indicizzazione della tabella basata sulle etichette numeriche degli elementi. Inoltre, non è necessario memorizzare l'etichetta di un elemento nella tabella, perché la posizione stessa la rappresenta. Tale tabella è chiamata **tabella ad accesso diretto**.

Se le etichette non sono numeriche, allora è ancora possibile usare un approccio ad accesso diretto. Chiameremo gli elementi $A[1]$, ... $A[N]$; ogni elemento $A[i]$ si compone di un'etichetta, o chiave, k_i e di un valore v_i. Si definisce quindi una funzione di traduzione $I(k)$ tale che $I(k)$ assume un valore compreso tra 1 e M per tutte le chiavi, e $I(k_i) \neq I(k_j)$ per qualunque i e j. In questo caso, si può ancora usare una tabella di lunghezza M ad accesso diretto.

I problemi con questo schema nascono quando $M >> N$. In questo caso, la quantità di celle della tabella che restano inutilizzate è grande, costituendo un utilizzo inefficiente della memoria. Un'alternativa sarebbe usare una tabella di lunghezza N e memorizzare gli N oggetti (eti-

chetta più valore) nelle N celle della tabella. Con questo schema, la quantità di memoria utilizzata è minima, però si ha un sovraccarico di elaborazione per effettuare delle ricerche nella tabella. Si hanno diverse possibilità:

- **Ricerca sequenziale:** questo approccio "forza bruta" è costoso in termini di tempo impiegato nel caso di tabelle grandi.
- **Ricerca associativa:** tramite hardware dedicato, tutti gli elementi di una tabelle possono essere ricercati simultaneamente. Non è un metodo generale né può essere applicato a tutte le possibili tabelle.
- **Ricerca binaria:** Supponendo che gli elementi presenti nella tabella siano ordinati in base alle etichette o alla loro traduzione numerica siano ordinati in modo crescente, allora una ricerca binaria è molto più veloce di una ricerca sequenziale (Tabella 8.6) e non richiede alcun hardware dedicato.

La ricerca binaria sembra essere promettente per la ricerca nelle tabelle. Il peggior difetto di questo approccio è rappresentato dal fatto che aggiungere nuovi elementi non è semplice, ed obbligherà a riordinare la tabella ad ogni inserimento. Pertanto, la ricerca binaria è utilizzata solamente con tabelle che possono ragionevolmente essere considerate statiche in quanto raramente subiscono variazioni.

L'ideale sarebbe evitare sia gli sprechi di memoria dovuti ad un approccio ad accesso diretto sia i costi computazionali aggiuntivi delle alternative precedentemente elencate. A tal fine, il metodo usato più frequentemente è l'**hashing**: sviluppato negli anni '50, è semplice da implementare ed ha due vantaggi. Innanzitutto permette di trovare la maggior parte degli elementi con un solo accesso, come nel caso dell'accesso diretto, inoltre è possibile inserire e cancellare elementi senza costi addizionali.

Tabella 8.6 *Lunghezza media di ricerca per uno degli N elementi in una tavola di lunghezza M*

Tecnica	Lunghezza della ricerca
Diretta	1
Sequenziale	$\frac{M+1}{2}$
Binaria	$\log_2 M$
Hash lineare	$\frac{2-\frac{N}{M}}{2-\frac{2N}{M}}$
Hash (overflow con concatenazione)	$1+\frac{N-1}{2M}$

La funzione di hashing può essere definita in base al seguente algoritmo supponendo che al massimo N elementi siano memorizzati in una tavola hash di lunghezza M, con $M \geq N$, ma non molto più grande di N. Per inserire un elemento nella tabella:

I1. Si converte l'etichetta dell'elemento da inserire in un numero pseudo-casuale n compreso tra 0 e M-1. Per esempio se l'etichetta è numerica, una funzione di traduzione comunemente adottata consiste nel dividere l'etichetta per M e considerare il resto della divisione come il valore di n.

I2. n è usato come indice per accedere ad una cella della tabella di hash.

a. se la cella corrispondente a n nella tabella è vuota, l'elemento (etichetta e valore) viene memorizzato in quella cella.

b. se la cella è già occupata, l'elemento viene memorizzato in un'area di overflow, come discusso nell'appendice.

Per effettuare la ricerca nella tabella di un elemento di cui si conosce l'etichetta:

R1. Si converte l'etichetta dell'elemento che si sta cercando in un numero n pseudo-casuale compreso tra 0 e M-1, usando la stessa funzione di traduzione usata per l'inserimento.

R2. n è usato come indice per accedere ad una cella della tabella di hash.

a. Se la cella corrispondente a n nella tabella è vuota, vuol dire che l'elemento non è stato precedentemente memorizzato nella tabella.

b. Se la cella è già occupata e le etichette corrispondono, allora il valore può essere recuperato.

c. Se la cella è già occupata e le etichette non corrispondono, si continua la ricerca nell'area di overflow.

Gli schemi di hashing si differenziano per il modo in cui gestiscono le collisioni. Un metodo comunemente adottato è la tecnica di **hashing lineare** usata nei compilatori. Con questo approccio la regola I2.b diventa

I2.b Se la cella è già occupata, si impone $n = n + 1 (\text{mod } M)$ e si torna al passo I2.a.

La regola R2.c viene modificata di conseguenza.

La Figura 8.24a è un esempio, in cui le etichette degli elementi da memorizzare sono numeriche e la tabella di hash ha otto celle ($M = 8$). La funzione di traduzione consiste nel prendere il resto della divisione per 8. La Figura suppone che gli elementi siano stati inseriti ordinati in modo crescente, sebbene questo non sia necessario. Perciò gli elementi 50 e 51 vengono assegnati alle celle 2 e 3 rispettivamente, e dal momento che queste sono vuote, vengono memorizzati. Anche l'elemento 74 viene assegnato alla cella 2, ma poiché la cella non è vuota, tenta l'inserimento nella cella 3. Anche la cella 3 è occupata, così finisce per inserirsi nella cella 4.

Non è facile determinare la lunghezza media della ricerca di un elemento in una tabella di hash aperta a causa dell'effetto di clustering. Una formula approssimativa è stata ottenuta da Schay e Spruth [SCHA62]:

$$\text{Lunghezza media di Ricerca} = \text{Formula: } \frac{2 - r}{2 - 2r}$$

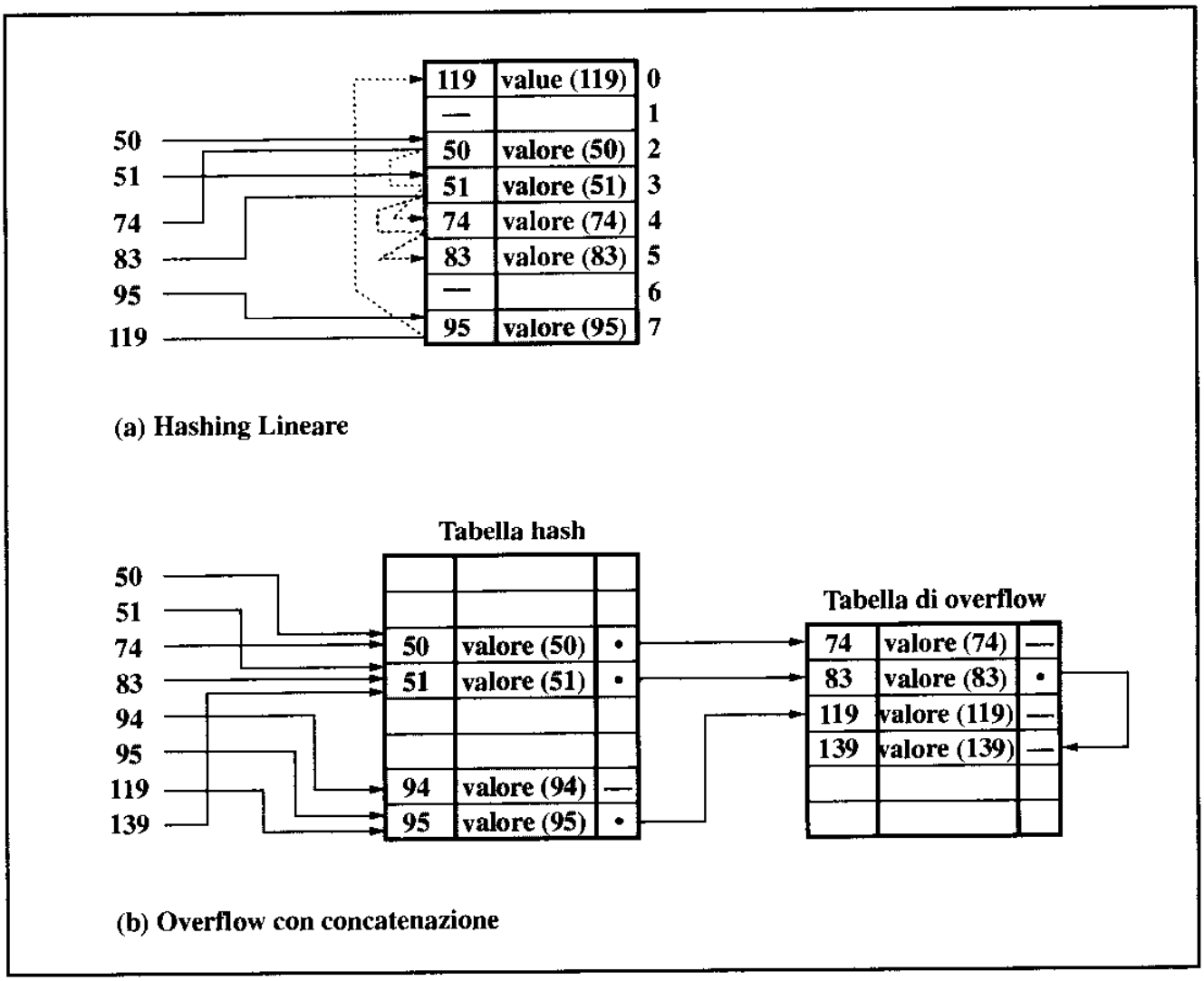


Figura 8.24 *Hashing*

dove $r = N/M$. Bisogna notare che il risultato è indipendente dalla dimensione della tabella poiché dipende soltanto da quanto è piena. Il risultato sorprendente è che in una tabella piena all'80%, il tempo medio di ricerca è ancora intorno a 3.

Però anche una tempo di ricerca di 3 può essere considerato lungo, inoltre una tabella di hash che usa una funzione di hashing lineare ha il problema della non facile cancellazione degli elementi. Un approccio più attraente, che fornisce tempi di ricerca più contenuti (Tabella 8.6) e consente le operazioni di cancellazione e inserimento, è la **gestione di overflow tramite catene**. Questa tecnica è illustrata nella Figura 8.24b. In questo caso c'è una tabella separata in cui sono inseriti gli elementi di overflow. Questa tabella contiene i puntatori che scorrono la catena delle celle associate alle posizioni della tabella di hash. In questo caso, la lunghezza media della ricerca, supponendo dati distribuiti casualmente, è:

$$\text{Lunghezza media di Ricerca} = 1 + \text{Formula}\ \frac{N-1}{2M}$$

Per grandi valori di N e M, questo valore si avvicina a 1.5 per $N = M$.
Perciò questa tecnica permette una memorizzazione compatta con ricerca rapida.

LO SCHEDULING

Parte 4

Il sistema operativo deve allocare le risorse del computer tra le potenziali necessità concorrenti di molti processi. Nel caso del processore, la risorsa da allocare è il tempo d'esecuzione sul processore e il mezzo di allocazione è lo scheduling. La funzione di scheduling deve essere progettata in modo da garantire la fairness, l'assenza di starvation dei processi, un uso efficiente del tempo di processore, un basso sovraccarico, tenendo in considerazione i diversi livelli di priorità o le scadenze in tempo reale per l'inizio o il completamento di certi processi.

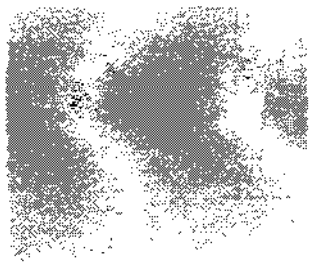
Negli anni passati, intense ricerche focalizzate sullo scheduling hanno portato all'implementazione di diversi algoritmi. Ai giorni nostri, l'enfasi nella ricerca sullo scheduling è rivolta verso lo sfruttamento dei sistemi multiprocessori, in modo particolare per le applicazioni multithread e lo scheduling in tempo reale.

Capitolo 9
Scheduling monoprocessore

Il Capitolo 9 si occupa dello scheduling su un sistema a singolo processore. In questo contesto limitato è possibile definire e chiarire molti problemi di progettazione relativi allo scheduling. Il capitolo comincia con una rassegna sui diversi livelli di scheduling e presenta e analizza alcuni algoritmi di scheduling.

Capitolo 10
Scheduling multiprocessore e in tempo reale

Il Capitolo 10 si rivolge ai punti focali della ricerca contemporanea sullo scheduling. La presenza di multiprocessori complica la decisione sul tipo di scheduling e apre la via a nuove opportunità. In particolare, con i multiprocessori è possibile schedulare simultaneamente per l'esecuzione thread multipli all'interno dello stesso processo. La prima parte



del Capitolo 10 è una panoramica sullo scheduling in sistemi multiprocessori e multithread. Il resto del capitolo è dedicato allo scheduling in tempo reale. I requisiti dell'esecuzione in tempo reale sono i più difficili da soddisfare per lo scheduler, perché vanno contro gli obiettivi di fairness e priorità, specificando limiti per l'inizio e la fine di un dato task o processo.

CAPITOLO 9

SCHEDULING MONOPROCESSORE

In un sistema in multiprogrammazione, i processi sono mantenuti nella memoria principale, e ogni processo alternativamente usa il processore o aspetta l'esecuzione dell'I/O o il completamento di un altro evento; infatti, il processore o i processori sono impegnati eseguendo un processo mentre gli altri processi sono in attesa.

Il punto centrale per la multiprogrammazione è lo scheduling: infatti, quattro tipi di scheduling sono generalmente coinvolti (Tabella 9.1). Uno di questi, l'I/O scheduling, è trattato più ampiamente nel Capitolo 11, dove l'I/O è approfondito. I tre tipi rimanenti di scheduling, che sono tipi di scheduling del processore, sono discussi in questo e nel successivo capitolo.

Il capitolo comincia con un esame dei tre tipi di scheduling del processore, mostrando come essi siano in relazione. Si può vedere che lo scheduling a lungo termine e lo scheduling a medio

Tabella 9.1 *Tipi di scheduling*

Scheduling a lungo termine	Si decide di aggiungere un processo all'insieme dei processi che devono essere eseguiti
Scheduling a medio termine	Si decide di aggiungere un processo all'insieme dei processi che sono parzialmente o completamente in memoria
Scheduling a breve termine	Si decide quale processo disponibile sarà eseguito dal processore
I/O scheduling	Si decide quale processo in attesa di I/O sarà gestito da un dispositivo di I/O disponibile

termine sono guidati principalmente da considerazioni sulle prestazioni relative al grado di multiprogrammazione. Questi argomenti sono stati trattati nel Capitolo 3, e più dettagliatamente nei Capitoli 7 e 8, perciò il resto di questo capitolo si concentra sullo scheduling a breve termine, in particolare nel caso di sistemi a singolo processore. Poiché l'uso di più processori aggiunge complessità all'argomento trattato, è meglio focalizzare l'attenzione sul caso di sistema monoprocessore, così da capire le differenze tra gli algoritmi di scheduling.

La Sezione 9.2 presenta i vari algoritmi che possono essere usati per realizzare lo scheduling a breve termine.

9.1 Tipi di scheduling

Lo scopo dello scheduling del processore è assegnare i processi che devono essere eseguiti al processore o ai processori, nel tempo, in modo da realizzare gli obiettivi del sistema, come il tempo di risposta, il throughput e l'efficienza del processore. In molti sistemi, quest'attività di scheduling è suddivisa in tre funzioni: scheduling a breve, medio o lungo termine. I nomi suggeriscono la frequenza relativa con cui le funzioni sono eseguite.

La Figura 9.1 collega le funzioni di scheduling al diagramma delle transizioni di stato del processo. Quando si crea un nuovo processo, si esegue lo scheduling a lungo termine: si decide di aggiungere un nuovo processo all'insieme dei processi che sono attivi al momento. Lo scheduling a medio termine è una parte della funzione di trasferimento su disco dei processi: si decide di aggiungere un processo a quelli che sono almeno parzialmente nella memoria princi-

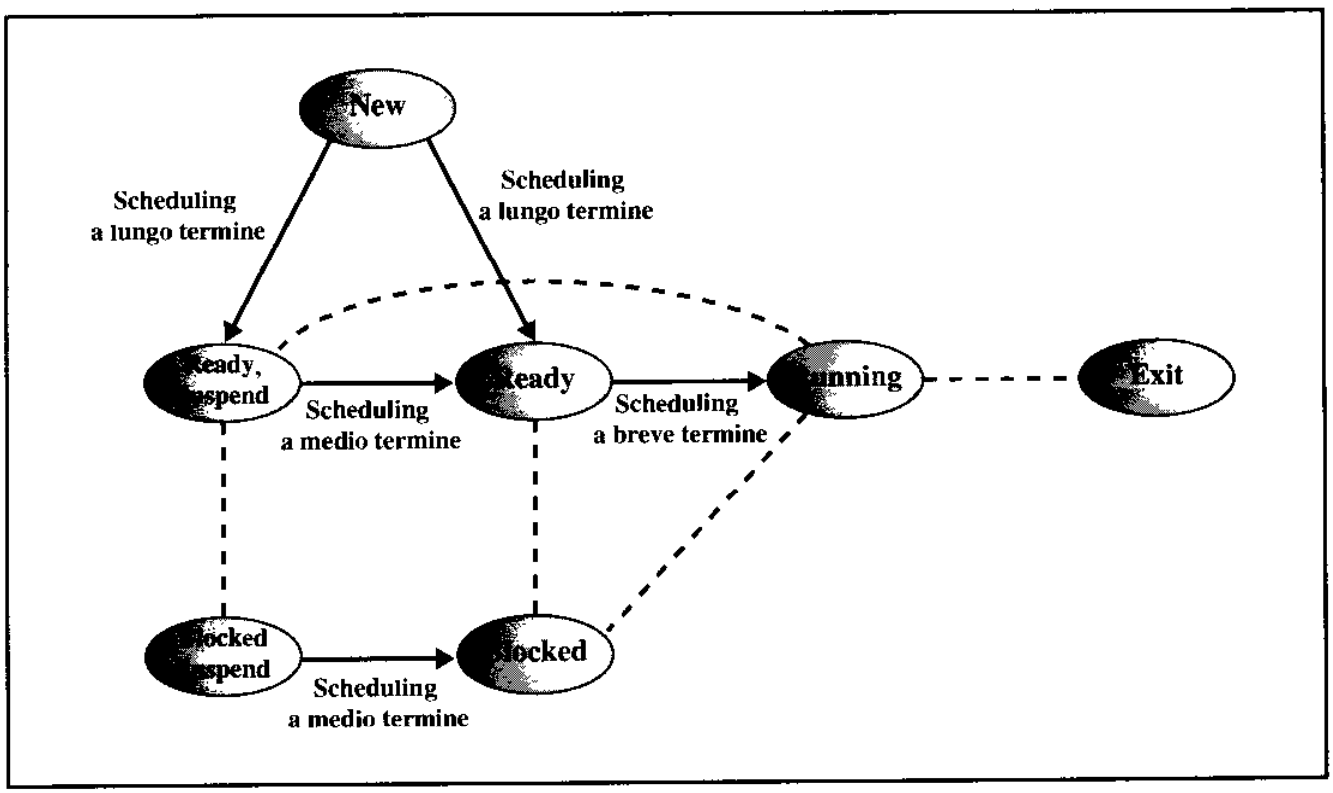


Figura 9.1 *Scheduling e transizioni di stato dei processi*

pale e perciò pronti ad essere eseguiti. Lo scheduling a breve termine decide quale processo Ready eseguire. La Figura 9.2 riorganizza il diagramma delle transizioni di stato per suggerire l'annidamento delle funzioni di scheduling.

Lo scheduling influenza le prestazioni del sistema perché determina i processi che aspetteranno e quelli che procederanno. Questo punto di vista è presentato nella Figura 9.3, che mostra le code coinvolte nelle transizioni di stato di un processo. Fondamentalmente, lo scheduling gestisce queste code in modo da minimizzare il tempo d'attesa in coda ed ottimizzare le prestazioni in un ambiente a code.

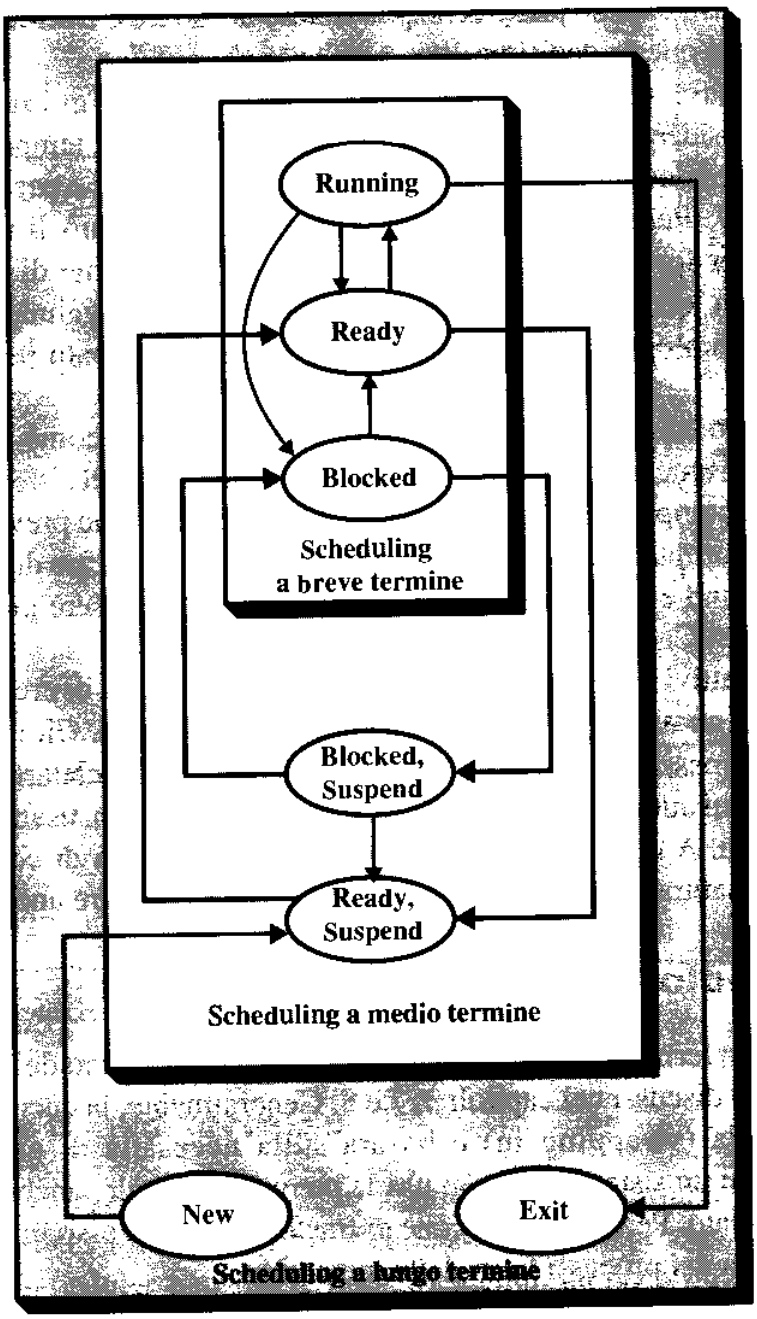


Figura 9.2 *Livelli di scheduling*

Scheduling a lungo termine

Lo scheduler a lungo termine determina quali programmi inserire nel sistema per l'esecuzione, perciò, controlla il grado di multiprogrammazione. Una volta inserito, un job, o un programma utente, diventa un processo e va ad aggiungersi alla coda dello scheduler a breve termine. In alcuni sistemi, un nuovo processo creato comincia in una condizione di trasferito su disco, nel qual caso è aggiunto alla coda dello scheduler a medio termine.

In un sistema batch, o per la parte batch di un sistema operativo generale, i nuovi job presentati sono diretti al disco e mantenuti in una coda batch. Lo scheduler a lungo termine preleva i processi dalla coda quando può; infatti, deve prendere due decisioni: se il sistema operativo può prendere uno o più processi, e quale job (anche più di uno) accettare e trasformarlo in processo. Consideriamo brevemente queste decisioni.

La decisione di creare un nuovo processo è generalmente guidata dal livello di multiprogrammazione desiderato. Più processi sono creati, minore è il tempo a disposizione di ogni processo per l'esecuzione (cioè, più processi sono in competizione per lo stesso tempo di utilizzo del processore). Perciò, lo scheduler a lungo termine può limitare il grado di multiprogrammazione per fornire un servizio soddisfacente all'insieme di processi che deve gestire. Ogni volta che un job termina, lo scheduler può decidere se aggiungere uno o più job, inoltre può essere chiamato ogni volta che il processore non è occupato per più di una certa percentuale di tempo.

Si può decidere che job ammettere con la semplice tecnica del first-come-first-served, oppure si può utilizzare uno strumento per la gestione delle prestazioni di un sistema. I criteri utilizzati nella scelta possono comprendere la priorità, il tempo di esecuzione previsto e i requisiti di I/O. Per esempio, se l'informazione è disponibile, lo scheduler può tentare di prendere un misto di processi processor-bound e I/O bound.[1] Inoltre la decisione può essere presa in dipendenza dalle risorse di I/O richieste, nel tentativo di bilanciare l'uso dell'I/O.

Per i programmi interattivi in un sistema time-sharing, una richiesta di creazione di processo è generata dall'azione di un utente che cerca di connettersi al sistema. Gli utenti time-sharing non sono semplicemente messi in coda in attesa che il sistema possa accettarli, il sistema operativo accetterà tutti gli utenti autorizzati fino a che il sistema non è saturo, usando qualche misura di saturazione predefinita. A quel punto, si risponde ad una richiesta di connessione con un messaggio che indica la saturazione del sistema e l'utente dovrà provare ancora.

Scheduling a medio termine

Lo scheduling a medio termine è parte della funzione di trasferimento dei processi su disco. I temi coinvolti sono stati discussi nei capitoli 3, 7 e 8. Generalmente, la decisione di introdurre un processo in memoria (swapping in) è basata sulla necessità di gestire il grado di multiprogrammazione. Su un sistema che non usa la memoria virtuale, anche la gestione della memoria è un tema correlato. Perciò, la decisione di trasferire un processo in memoria considererà i requisiti di memoria dei processi fuori dalla memoria.

---

[1] Un processo è detto processor bound se esegue prevalentemente lavoro computazionale e se usa solo occasionalmente i dispositivi di I/O. Un processo è detto I/O bound se passa la maggior parte del tempo usando i dispositivi di I/O piuttosto che il processore.

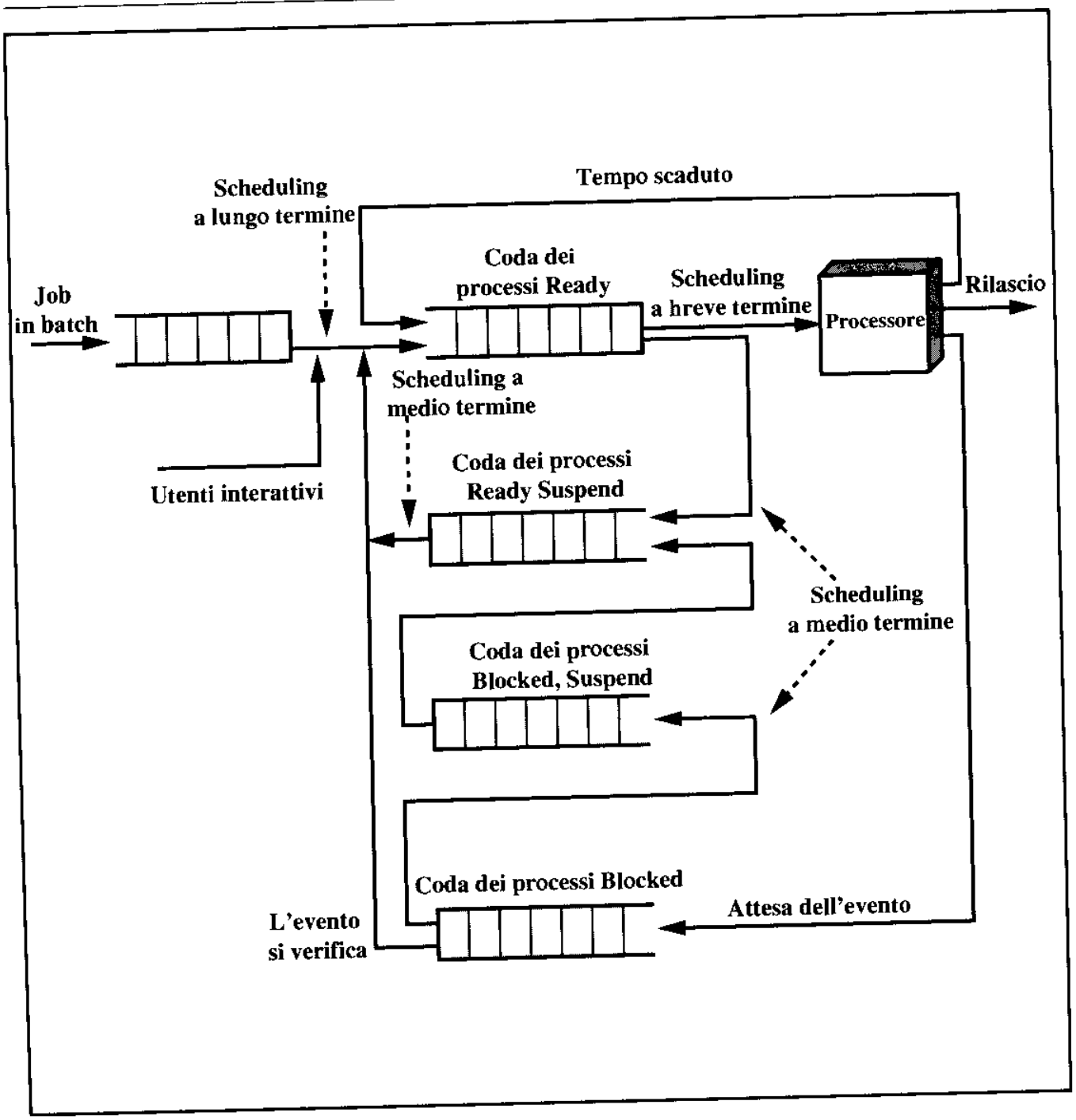


Figura 9.3 *Diagramma delle code usate per lo scheduling*

Scheduling a breve termine

In termini di frequenze di esecuzione, lo scheduler a lungo termine è eseguito poco frequentemente e prende la decisione grossolana se caricare un nuovo processo in memoria e, nel caso, quale caricare. Lo scheduler a medio termine è eseguito un poco più spesso per decidere se è necessario un trasferimento su disco. Lo scheduler a breve termine, detto anche allocatore o dispatcher, è eseguito più frequentemente e prende la decisione più fine di quale processo eseguire per prossimo.

Lo scheduler a breve termine è chiamato ogni volta che si verifica un evento che può portare alla sospensione del processo corrente o che consente di prerilasciare il processo attualmente in esecuzione in favore di un altro. Di seguito, alcuni esempi di tali eventi:

- Interruzioni di clock
- Interruzioni di I/O
- Chiamate a sistema operativo
- Segnali.

9.2 Algoritmi di scheduling

I criteri dello scheduling a breve termine

Generalmente, si valutano le strategie di scheduling sulla base dei seguenti criteri.

I criteri solitamente usati possono essere classificati in due dimensioni: la prima consiste nel fare una distinzione tra criteri orientati agli utenti e criteri orientati al sistema. I criteri **orientati all'utente** sono relativi a come il singolo utente o processo percepisce il comportamento del sistema. Un esempio è il tempo di risposta in un sistema interattivo, cioè il tempo trascorso tra l'invio di una richiesta e l'inizio dell'uscita della risposta. Questa quantità è visibile all'utente e naturalmente gli interessa. Si vorrebbe una strategia di scheduling che fornisca un buon servizio ai vari utenti: nel caso del tempo di risposta, si può definire una soglia (ad esempio, 2 secondi). Lo scopo del meccanismo di scheduling consiste nel massimizzare il numero di utenti che sperimentano un tempo medio di risposta di non più di 2 secondi.

Gli altri criteri sono **orientati al sistema** il cui punto focale sta nell'efficiente ed efficace utilizzazione del processore. Un esempio è il throughput, che è la frequenza di completamento dei processi. Questa è certamente una valida misura delle prestazioni del sistema, ed è una misura che deve essere massimizzata; però si concentra sulle prestazioni del sistema piuttosto che sui servizi prestati all'utente: perciò, è di competenza dell'amministratore di sistema ma non degli utenti.

Mentre i criteri orientati all'utente sono praticamente importanti su tutti i sistemi, i criteri orientati al sistema sono generalmente di importanza minore sui sistemi a singolo utente, dove non è importante raggiungere un'alta utilizzazione del processore o un alto throughput fintanto che la prontezza di risposta del sistema alle applicazioni dell'utente è accettabile.

Un'altra dimensione di classificazione consiste nel distinguere i criteri che sono in relazione con le prestazioni da quelli che non sono in diretta relazione con essa. I criteri che sono **relativi alle prestazioni** sono quantitativi e generalmente possono essere prontamente misurati. Alcuni esempi sono il tempo di risposta e il throughput. I criteri **non relativi alle prestazioni** o sono qualitativi, o non sono facilmente misurabili e analizzabili: un esempio di tali criteri è la prevedibilità. Si vorrebbe che alcune caratteristiche dei servizi forniti agli utenti siano indipendenti dagli altri lavori eseguiti dal sistema; in una qualche misura, questo criterio può essere misurato, calcolando le variazioni come funzione del carico di lavoro. Comunque, questo non è

diretto quanto misurare il throughput o il tempo di risposta come funzione del carico di lavoro.

La Tabella 9.2 riassume i criteri chiave dello scheduling. Questi sono interdipendenti e non è possibile ottimizzarli tutti allo stesso tempo. Per esempio, fornire un buon tempo di risposta può richiedere un algoritmo di scheduling che cambia frequentemente i processi, ma questo aumenta l'overhead nel sistema, riducendo il throughput. Perciò la progettazione di una strategia di

Tabella 9.2 *Criteri di scheduling*

	Orientati all'utente, Relativi alle prestazioni
Tempo di risposta	Per un processo interattivo, questo è il tempo che trascorre dall'invio di una richiesta fino a quando la risposta non comincia ad essere ricevuta. Spesso un processo può cominciare a produrre qualche output per l'utente, mentre continua l'esecuzione. Perciò, dal punto di vista dell'utente, questa è una misura migliore del tempo di turnaround. La disciplina di scheduling dovrebbe tentare di raggiungere bassi tempi di risposta e di massimizzare il numero di utenti interattivi che hanno un tempo di risposta accettabile.
Tempo di turnaround	È l'intervallo di tempo tra l'invio di un processo e il suo completamento. Comprende il tempo di esecuzione e il tempo speso in attesa delle risorse, compreso il processore. Questa è una misura appropriata per i job batch.
Scadenze	Quando si può specificare il termine di completamento di un processo, la disciplina di scheduling può subordinare altri obiettivi oltre a quello di massimizzare la percentuale di scadenze raggiunta.
	Orientati all'utente, Altri
Prevedibilità	Un dato job può essere eseguito nello stesso tempo e allo stesso costo, indipendentemente dal carico del sistema. Un'ampia variazione nel tempo di risposta o nel tempo di turnaround è fastidiosa per l'utente: potrebbe segnalare un'ampia oscillazione nel carico di lavoro del sistema oppure la necessità di adattare il sistema per evitare l'instabilità
	Orientati al sistema, Relativi alle prestazioni
Throughput	La strategia di scheduling dovrebbe tentare di massimizzare il numero di processi completati per unità di tempo. Così si misura la quantità di lavoro eseguito, che dipende chiaramente dalla lunghezza media di un processo, ma che è anche influenzata dalla strategia di scheduling che può avere effetti sull'utilizzazione
Utilizzazione del processore	Questa è la percentuale del tempo in cui il processore è occupato. Per un sistema costoso e condiviso, questo è un criterio significativo, mentre in sistemi a singolo utente e in alcuni altri sistemi, come i sistemi a tempo reale, questo criterio è meno importante.
	Orientati al sistema, Altri
Fairness	In assenza di indicazioni da parte dell'utente, oppure del sistema, i processi devono essere trattati allo stesso modo, e nessun processo deve subire starvation.
Priorità	Quando si assegna una priorità ai processi, la strategia di scheduling dovrebbe favorire i processi a più alta priorità.
Bilanciamento delle risorse	La strategia di scheduling deve tenere impegnate le risorse del sistema, favorendo i processi che sottoutilizzano le risorse più impegnate. Questo criterio coinvolge anche lo scheduling a medio e lungo termine.

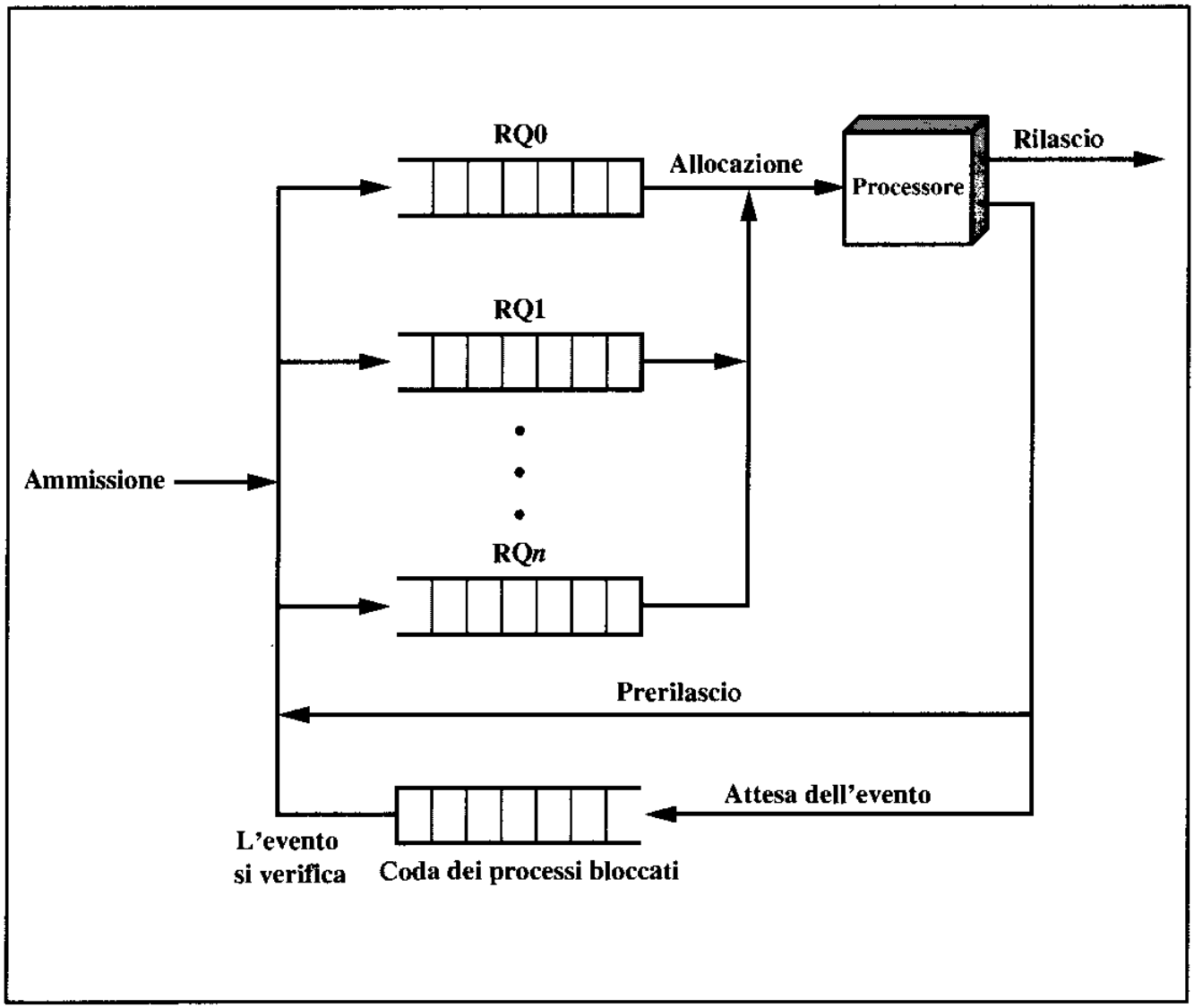


Figura 9.4 *Code di priorità*

scheduling richiede un compromesso tra i requisiti in competizione; i pesi relativi dati ai vari requisiti dipenderanno dalla natura e dall'uso del sistema.

Nella maggior parte dei sistemi operativi interattivi, siano essi a singolo utente o in time-sharing, il requisito critico è il tempo di risposta adeguato. Per l'importanza di questo requisito, e poiché la definizione di adeguatezza varierà da un'applicazione all'altra, l'argomento è approfondito nell'appendice di questo capitolo.

L'uso delle priorità

In molti sistemi, ad ogni processo è assegnata una priorità e lo scheduler deve sempre scegliere un processo a più alta priorità a discapito di processi con più bassa priorità. La Figura 9.4 illustra l'uso delle priorità. Per chiarezza, il diagramma a code è semplificato, ignorando l'esistenza di più code di processi bloccati e di stati Suspended (confrontare con la Figura 3.6a). Invece di una sola coda di processi Ready, si usa un insieme di code, in ordine di priorità discen-

dente: RQ0, RQ1, ..., RQn, con priorità [RQi] > priorità [RQj] per $i < j$. Quando deve essere fatta una selezione di scheduling, lo scheduler comincerà dalla coda di processi Ready con priorità più alta (RQ0): se ci sono uno o più processi nella coda, una strategia di scheduling ne seleziona uno, se, invece, RQ0 è vuota, allora si esamina RQ1 e così via.

Il problema con uno schema di scheduling basato solamente sulla priorità è che i processi con la priorità più bassa possono soffrire di starvation. Questo succederà se sono presenti stabilmente processi Ready a più alta priorità. Per superare questo problema, la priorità di un processo può cambiare in dipendenza del tempo di presenza in una coda o del corso della sua esecuzione. In seguito sarà presentato un esempio di ciò.

Strategie di scheduling alternative

La Tabella 9.3 presenta alcune informazioni riassuntive sulle varie strategie di scheduling esaminate in questa sezione. La **funzione di selezione** determina quale processo, tra i processi Ready, selezionare per la prossima esecuzione. La funzione può basarsi sulla priorità, sulle richieste di risorsa o sulle caratteristiche di esecuzione del processo. Nell'ultimo caso, tre sono le quantità significative:

w = tempo fin qui speso nel sistema, aspettando e eseguendo

e = tempo speso fin qui in esecuzione

s = tempo totale di servizio richiesto dal processo, compreso e

Per esempio, la funzione di selezione max[w] indica la strategia del first-come-first-served (FCFS).

Il **modo di decisione** specifica gli istanti in cui si esegue la funzione di selezione. Ci sono due categorie generali:

- **Senza prerilascio:** in questo caso, una volta che il processo è in esecuzione, continua la sua esecuzione fino alla terminazione o finché si blocca in attesa di I/O o se richiede altri servizi del sistema operativo.
- **Con prerilascio:** Il processo al momento in esecuzione può essere interrotto e spostato in stato Ready dal sistema operativo. La decisione di prerilascio può essere presa quando arriva un nuovo processo, quando avviene un interrupt che trasferisce un processo dallo stato Blocked allo stato Ready, oppure periodicamente in base all'interruzione di clock.

Le strategie con prerilascio provocano un overhead maggiore di quelle senza prerilascio, ma possono fornire un servizio migliore all'intero insieme dei processi, perché impediscono ad ogni processo di monopolizzare il processore per molto tempo. Inoltre, il costo del prerilascio può essere mantenuto relativamente basso usando meccanismi efficienti per il cambio di processo (si ricerca l'aiuto maggiore possibile da parte dell'hardware) e fornendo tanta memoria principale da contenere una gran percentuale dei programmi.

Per descrivere le varie strategie di scheduling, si userà il seguente insieme di processi come esempi:

Tabella 9.3 *Caratteristiche delle varie strategie di scheduling*

	FCFS	Round Robin	SPN	SRT	HRRN	Feedback
Funzione di selezione	max[w]	costante	min[s]	min[s - e]	max $\left(\frac{w+s}{s}\right)$	(vedere il testo)
Modalità di decisione	Senza prerilascio	Con prerilascio (allo scadere del quanto di tempo)	Senza prerilascio	Con prerilascio (all'arrivo)	Senza prerilascio	Con prerilascio (allo scadere del quanto di tempo)
Throughput	Non enfatizzato	Può essere basso se il quanto è troppo breve	Alto	Alto	Alto	Non enfatizzato
Tempo di risposta	Può essere alto, specialmente se c'è una grande varianza nel tempo di esecuzione del processo	Fornisce un buon tempo di risposta per i processi brevi	Fornisce un buon tempo di risposta per i processi brevi	Fornisce un buon tempo di risposta	Fornisce un buon tempo di risposta	Non enfatizzato
Overhead	Minimo	Basso	Può essere alto	Può essere alto	Può essere alto	Può essere alto
Effetto sui processi	Penalizza i processi brevi e quelli I/O bound	Trattamento fair	Penalizza i processi lunghi	Penalizza i processi lunghi	Buon bilanciamento	Può favorire i processi I/O bound
Starvation	No	No	Possibile	Possibile	No	Possibile

w = tempo trascorso complessivamente nel sistema, in attesa e in esecuzione

e = tempo trascorso complessivamente in esecuzione

s = tempo di servizio totale richiesto dal processo, compreso e.

Processo	Tempo di arrivo	Tempo di servizio
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Si supponga di considerarli job in batch, e si supponga che il tempo di servizio sia il tempo totale richiesto per l'esecuzione. Alternativamente, è possibile considerare i processi elencati come processi in esecuzione che richiedono un uso alternato del processore e dell'I/O ripetutamente; in quest'ultimo caso, i tempi di servizio rappresentano il tempo di processore richiesto per un ciclo. In entrambi i casi, in termini di modello a code (vedere appendice A), questa quantità corrisponde al tempo di servizio.

First Come First Served

La più semplice strategia di scheduling è il first-come-first-served (primo arrivato primo servito - FCFS), o first-in-first-out (FIFO). Non appena un processo diventa Ready, s'inserisce nella coda dei processi Ready. Quando il processo al momento in esecuzione si sospende, il più vecchio processo nella coda dei processi Ready è selezionato per l'esecuzione.

La Figura 9.5 mostra il cammino di esecuzione del nostro esempio per un ciclo, e la Tabella 9.4 fornisce alcuni risultati chiave. Dapprima si determina il tempo a cui si sospende un processo, dal quale si può determinare il tempo di turnaround. Nei termini del modello a code, questo è il tempo che il processo trascorre in coda, o il tempo totale che un elemento trascorre nel sistema (tempo di attesa più tempo di servizio). Un numero più utile è il tempo di turnaround normalizzato, che è il rapporto tra il tempo di turnaround e il tempo di servizio. Questo valore indica il ritardo relativo subito da un processo. Tipicamente, più lungo è il tempo di esecuzione di un processo, maggiore è il ritardo assoluto totale che può essere tollerato. Il valore minimo di questo rapporto è 1.0; aumentare i valori significherebbe decrescere il livello del servizio.
FCFS offre migliori prestazioni per i processi lunghi che per quelli brevi. Si considerino i seguenti esempi, basati su un esempio di [FINK88]:

Processo	Tempo di arrivo	Tempo di servizio (T_s)	Tempo di inizio	Tempo di fine	Tempo di turnaround (T_q)	T_q/T_s
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99
Media					100	26

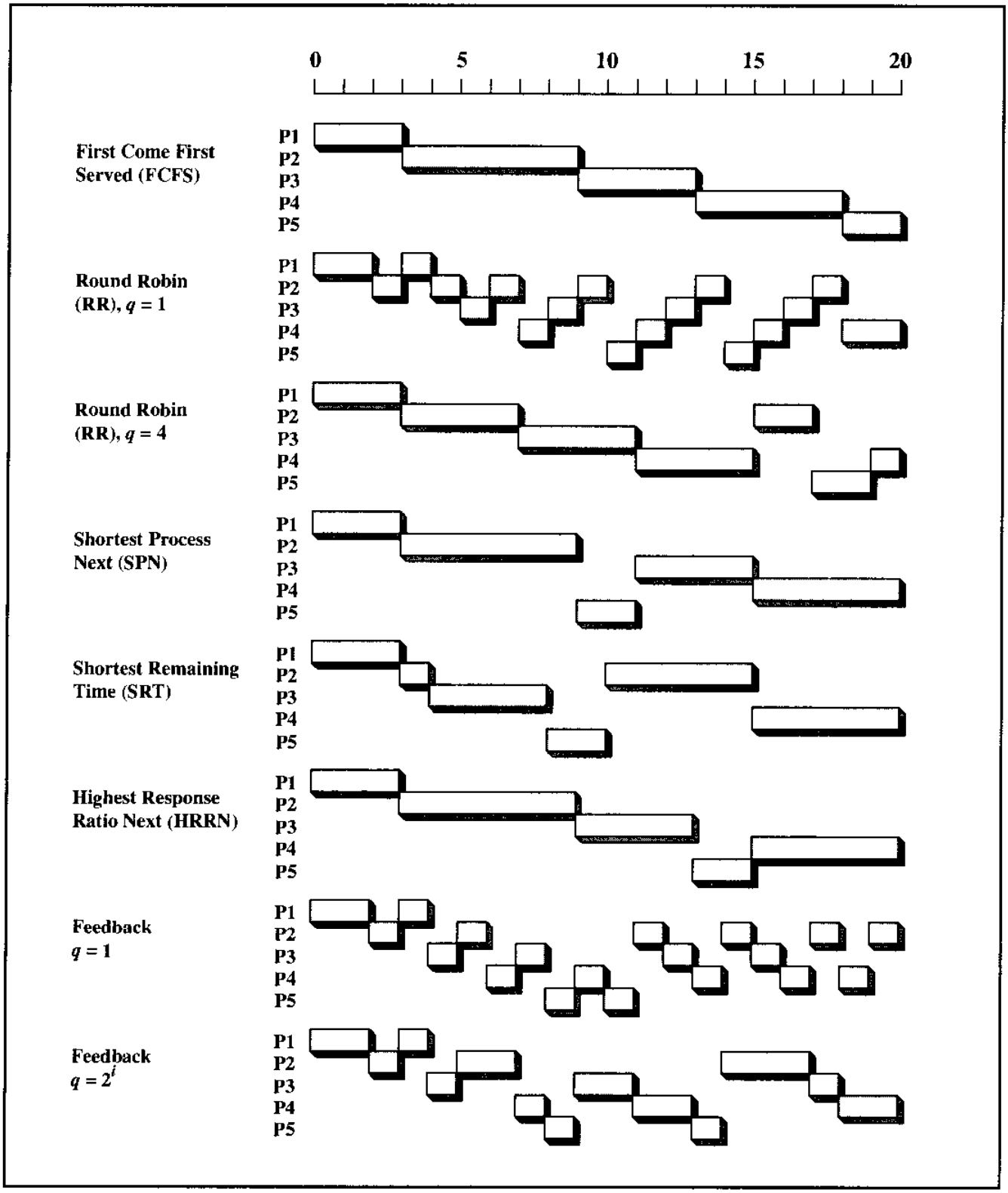


Figura 9.5 *Confronto tra le strategie di scheduling*

Il tempo di turnaround normalizzato per il processo C è intollerabile: il tempo totale che trascorre nel sistema è 100 volte il tempo di esecuzione richiesto, e questo accade ogni volta che un processo breve arriva subito dopo un processo lungo. D'altra parte, anche in quest'esempio estremo, i processi lunghi non vanno male: il processo D ha un tempo di turnaround che è quasi

Tabella 9.4 *Confronto tra le strategie di scheduling*

	Processo	1	2	3	4	5	Media
	Tempo di arrivo	0	2	4	6	8	
	Tempo di servizio (T_s)	3	6	4	5	2	
FCFS	Tempo di fine	3	9	13	18	20	
	Tempo di turnaround (T_q)	3	7	9	12	12	8.60
	T_q/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR q = 1	Tempo di fine	4	18	17	20	15	
	Tempo di turnaround (T_q)	4	16	13	14	7	10.80
	T_q/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR q = 4	Tempo di fine	3	17	11	20	19	
	Tempo di turnaround (T_q)	3	15	7	14	11	10.00
	T_q/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN	Tempo di fine	3	9	15	20	11	
	Tempo di turnaround (T_q)	3	7	11	14	3	7.6
	T_q/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	Tempo di fine	3	15	8	20	10	
	Tempo di turnaround (T_q)	3	13	4	14	2	7.20
	T_q/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	Tempo di fine	3	9	13	20	15	
	Tempo di turnaround (T_q)	3	7	9	14	7	8.00
	T_q/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB q = 1	Tempo di fine	4	20	16	19	11	
	Tempo di turnaround (T_q)	4	18	12	13	3	10.00
	T_q/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB q = $2^{(i-1)}$	Tempo di fine	4	17	18	20	14	
	Tempo di turnaround (T_q)	4	15	14	14	6	10.60
	T_q/T_s	1.33	2.50	3.50	2.80	3.00	2.63

il doppio di quello di C, ma il suo tempo di attesa normalizzato è inferiore a 2.0.

Il problema di questa strategia consiste nel fatto che tende a favorire i processi processor-bound a discapito di quelli I/O-bound. Si consideri un insieme di processi, uno dei quali usa prevalentemente il processore (processor bound) mentre gli altri preferiscono l'I/O (I/O bound). Quando un processo processor-bound è in esecuzione, tutti i processi I/O-bound devono aspettare; alcuni di questi possono essere nelle code di I/O (in stato Blocked) ma possono tornare nella coda dei processi Ready mentre il processo processor-bound è in esecuzione. A questo punto, la maggior parte dei dispositivi di I/O può essere in ozio, anche se c'è del lavoro potenziale per loro. Quando il processo al momento in esecuzione esce dallo stato Running, i processi I/O-bound si spostano rapidamente in stato Running, ma si bloccano sui rispettivi eventi di I/O. Se anche il processo processor-bound è bloccato, il processore rimane in ozio. Perciò, FCFS può usare inefficientemente sia il processore sia i dispositivi di I/O.

FCFS non è di per se stesso un'alternativa attraente per un sistema a singolo processore, ma è spesso combinato con uno schema di priorità per fornire uno scheduler efficiente. Pertanto, lo scheduler può mantenere alcune code, una per ogni livello di priorità, e distribuire i processi all'interno di ogni coda utilizzando il first-come-first-served. Un esempio di tale sistema si vedrà in seguito, discutendo dello scheduling con feedback.

Round robin

Un semplice modo per ridurre le penalità che i processi brevi subiscono con FCFS, è usare il prerilascio basato sul clock. La più semplice di tali strategie è il round robin. Un'interruzione di clock è generata ad intervalli periodici. Quando avviene un'interruzione, il processo in esecuzione al momento è inserito nella coda dei processi Ready e il successivo job Ready è selezionato con FCFS. Questa tecnica è anche conosciuta con il nome di **time slicing**, perché ad ogni processo si dà un po' di tempo prima di essere prerilasciato.

Con il round robin, la principale preoccupazione in fase di progettazione è la lunghezza del quanto di tempo, o slice, da usare. Se il quanto di tempo è molto breve, i processi brevi passeranno abbastanza velocemente all'interno del sistema. D'altra parte, c'è un tempo aggiuntivo di elaborazione dovuto alla gestione dell'interruzione di clock ed all'esecuzione delle funzioni di scheduling ed allocazione, perciò devono essere evitati quanti di tempo troppo brevi. Un criterio utile è di avere un quanto di tempo leggermente maggiore del tempo richiesto per una tipica interazione: se fosse minore, la maggior parte dei processi richiederebbe almeno due quanti di tempo. La Figura 9.6 illustra gli effetti che questo ha sul tempo di risposta: notare che nel caso limite di un quanto di tempo più lungo del più lungo processo Running, il round robin degenera in FCFS.

La Figura 9.5 e la Tabella 9.4 mostrano i risultati dell'esempio precedente usando un quanto di tempo q di 1 e 4 unità di tempo, rispettivamente. Bisogna notare che il processo 5, che è il job più breve, gode di un significativo miglioramento per un quanto di tempo di un'unità.

Il round robin è particolarmente efficiente in un sistema generale time-sharing, o in un sistema orientato alle transazioni; un inconveniente del round robin è il trattamento che riserva ai processi processor-bound e I/O-bound. Generalmente, un processo I/O-bound ha un burst di processo (tempo speso in esecuzione fra due operazioni di I/O) minore di quello di un processo processor-bound. Se ci sono sia processi I/O-bound sia processi processor-bound, un processo I/O-bound userà il processore per un breve tempo, poi sarà bloccato per l'I/O; aspetterà il completamento dell'operazione di I/O e poi s'inserirà nella coda dei processi Ready. Invece, un processo processor-bound generalmente usa un intero quanto di tempo e poi torna nella coda dei processi Ready. Perciò, i processi processor-bound tendono a ricevere una porzione del tempo di processore non equa, il che provoca scarse prestazioni dei processi I/O-bound, un uso inefficiente dei dispositivi di I/O e un aumento della varianza del tempo di risposta.

[HAL91] suggerisce un miglioramento del round robin, chiamato round robin virtuale (VRR), che elimina questa scorrettezza. La Figura 9.7 illustra lo schema: i nuovi processi arrivano e sono inseriti nella coda dei processi Ready, che è gestita con FCFS. Quando scade il tempo del processo in esecuzione, esso è reinserito nella coda dei processi Ready. Quando un processo è bloccato per l'I/O, s'inserisce in una coda di processi in attesa dell'I/O: finora tutto è come al solito. La nuove caratteristica è una coda ausiliaria FCFS nella quale sono inseriti i processi,

dopo essere stati rilasciati da un blocco di I/O. Quando si deve decidere un'allocazione dei processi, i processi che sono nella coda ausiliaria hanno la precedenza su quelli della coda dei processi Ready. Quando un processo è prelevato dalla coda ausiliaria, viene eseguito per un quanto, lungo al più come il quanto di tempo di base, meno il tempo totale trascorso in esecuzione dall'ultima volta che è stato selezionato dalla coda principale dei processi Ready. Studi sulle prestazioni condotti dagli autori indicano che quest'approccio è davvero superiore, in termini di fairness, al round robin.

Shortest Process Next

Un altro approccio per ridurre il vantaggio in favore dei processi lunghi, intrinseco per FCFS, è la strategia Shortest Process Next (SPN: il prossimo processo è il più breve). Questa è una strategia senza prerilascio, nella quale si sceglie come successivo processo da eseguire il proces-

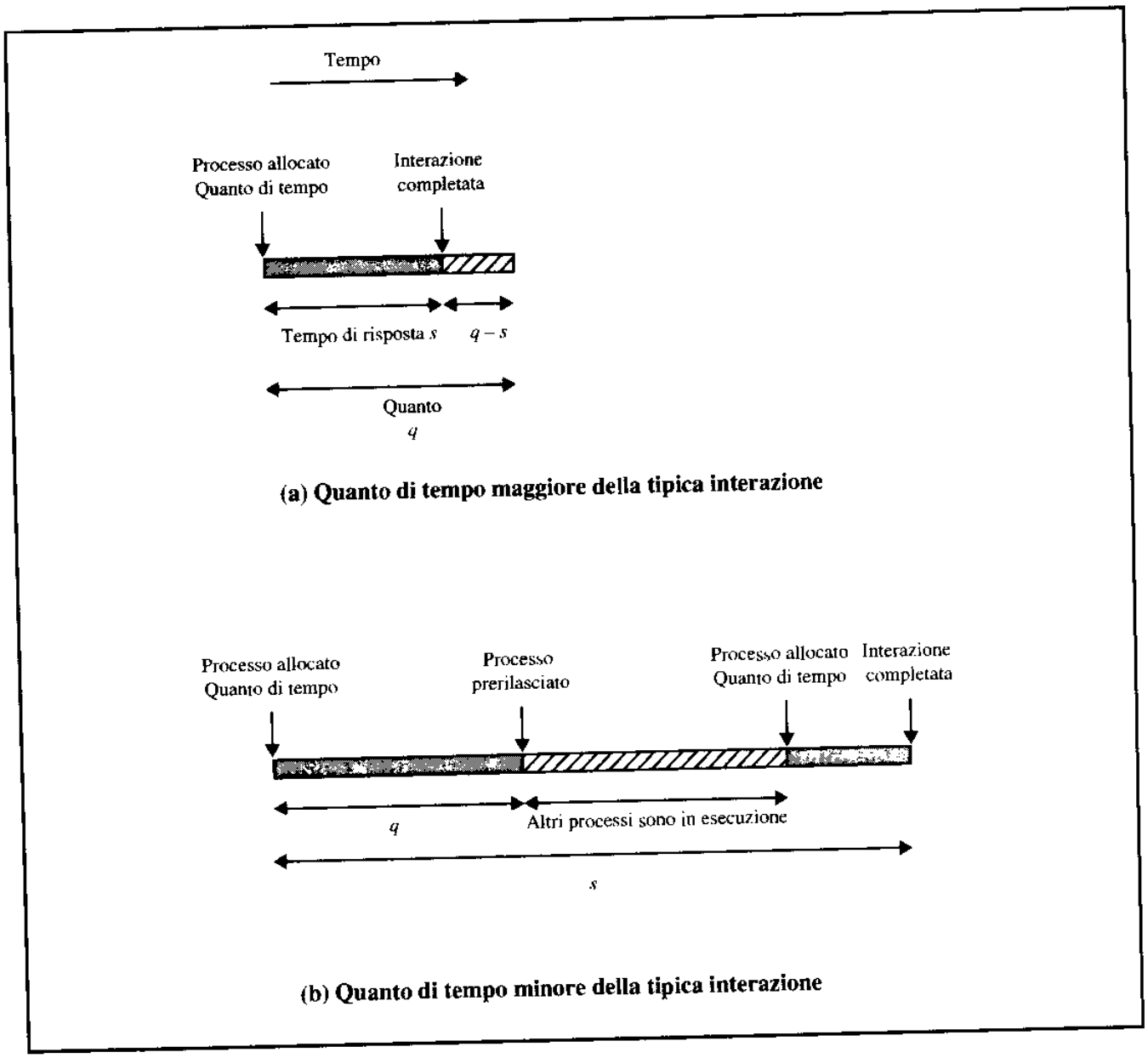


Figura 9.6 *Effetti della dimensione del quanto di tempo di prerilascio*

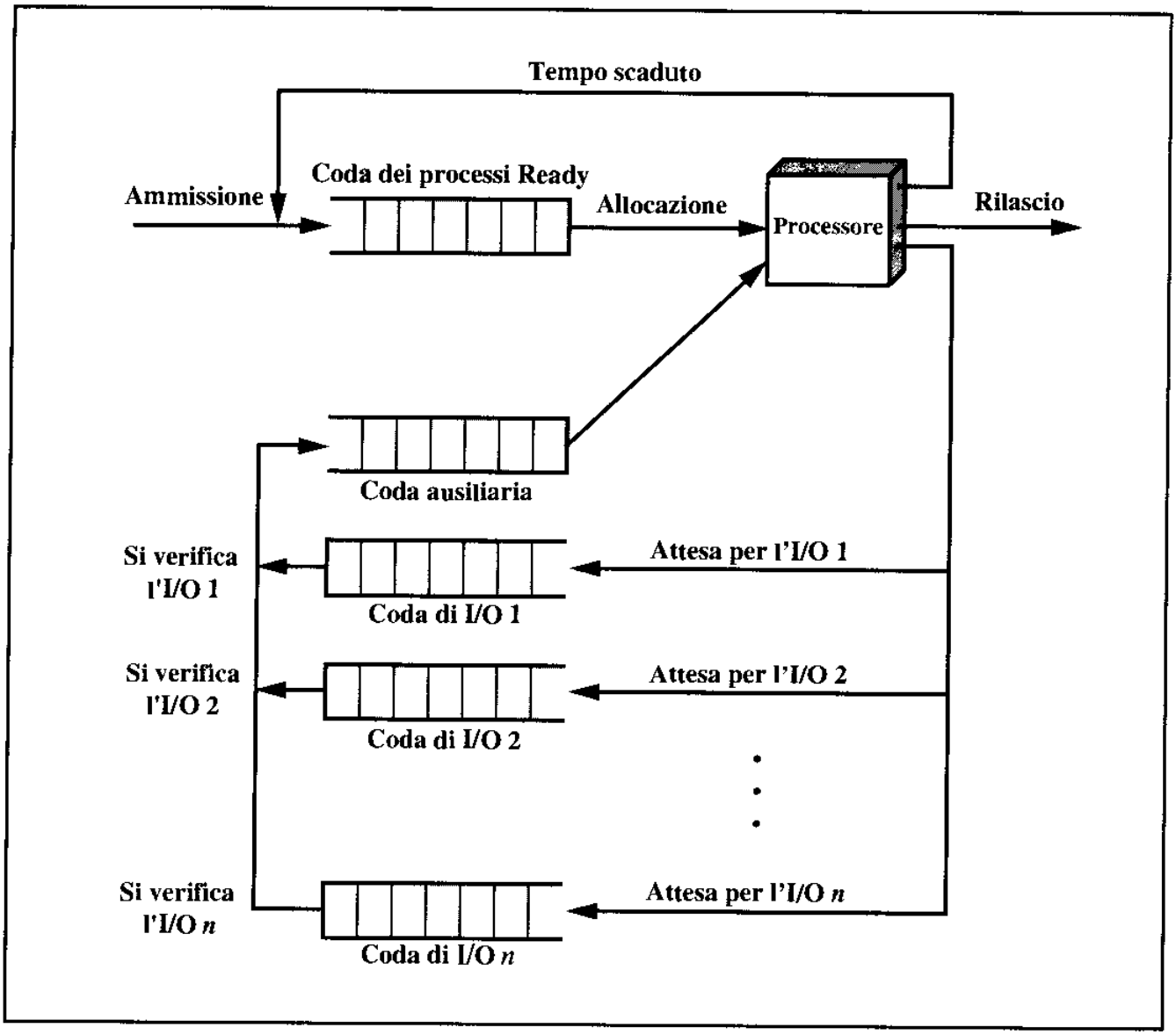


Figura 9.7 *Diagramma a code per lo scheduler round robin virtuale*

so con il minor tempo esecuzione previsto. Perciò un processo breve balzerà in testa alla coda sorpassando i job più lunghi.

La Figura 9.5 e la Tabella 9.4 mostrano i risultati per l'esempio considerato. Bisogna notare che il processo 5 è servito molto prima di quando si è usato FCFS, e le prestazioni complessive sono significativamente migliorate in termini di tempo di risposta. Comunque, la variabilità dei tempi di risposta aumenta, specialmente per i processi più lunghi, e perciò la prevedibilità è ridotta.

Una difficoltà che sorge nell'uso della strategia SPN è la necessità di conoscere, o almeno stimare, il tempo di esecuzione previsto per ogni processo. Per i job in batch, il sistema può richiedere al programmatore di stimare il valore e fornirlo al sistema operativo; se la stima del programmatore è decisamente inferiore al tempo di esecuzione effettivo, il sistema può "abortire" il processo. In un ambiente di produzione, gli stessi job girano frequentemente e si possono fare statistiche. Per i processi interattivi, il sistema operativo può calcolare una media corrente ad ogni burst per ogni processo. La seguente formula permette di fare il calcolo più semplice:

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i \tag{9.1}$$

dove

T_i = tempo di esecuzione del processore per la i-esima istanza di questo processo (tempo totale di esecuzione per job in batch; tempo di un burst di processo per job interattivo)

S_i = valore previsto per la i-esima istanza

S_1 = valore previsto per la prima istanza; non calcolato

Per evitare di ricalcolare l'intera somma ogni volta, si può riscrivere la formula precedente come:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n \tag{9.2}$$

Bisogna notare che questa formula dà uguale peso ad ogni istanza; ma tipicamente, si vorrà dare più peso alle istanze più recenti, perché prevedono più verosimilmente il comportamento futuro. Una tecnica comune per predire un valore futuro sulle basi di una serie temporale dei valori passati è la *media esponenziale*:

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n \tag{9.3}$$

dove α è un peso costante ($0 < \alpha < 1$) che determina il peso relativo dato alle osservazioni più e meno recenti. Confrontando questa equazione con la (9.2), con un valore costante di α, indipendentemente dal numero delle osservazioni passate, tutti i valori passati sono considerati, ma i più distanti hanno meno peso. Per vedere questo più chiaramente, si consideri la seguente espansione dell'Equazione (9.3):

$$S_{n+1} = \alpha T_n + (1-\alpha)\,\alpha T_{n-1} + \ldots + (1-\alpha)^i\,\alpha T_{n-i} + (1-\alpha)^n S_1 \tag{9.4}$$

Poiché sia α che $(1-\alpha)$ sono minori di uno, ogni termine successivo nell'equazione (9.4) è sempre più piccolo. Per esempio, per $\alpha = 0.8$, l'Equazione (9.4) diventa

$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \ldots$$

Più vecchia è l'osservazione, meno conta nella media.

La dimensione del coefficiente, come funzione della sua posizione nell'espansione, è mostrata in Figura 9.8. Più grande è il valore di α, maggiore è il peso dato alle osservazioni più recenti: per $\alpha = 0.8$, virtualmente tutto il peso è dato alle quattro osservazioni più recenti, mentre per $\alpha = 0.2$, la media considera le otto osservazioni più recenti. Il vantaggio di usare un valore di α vicino ad uno è che la media rifletterà un rapido cambio nella quantità osservata; lo svantaggio è che se c'è un breve sbalzo nel valore della quantità osservata, che poi si fissa su qualche valore medio, l'uso di un grande valore di a risulterà in cambiamenti a balzi della media.

La Figura 9.9 confronta la media semplice con la media esponenziale (per due diversi valori di α). Nella Figura 9.9a il valore osservato parte da 1, cresce gradualmente fino a 10 e poi si ferma, mentre nella Figura 9.9b, parte da 20, scende gradualmente a 10 e poi si ferma; in entram-

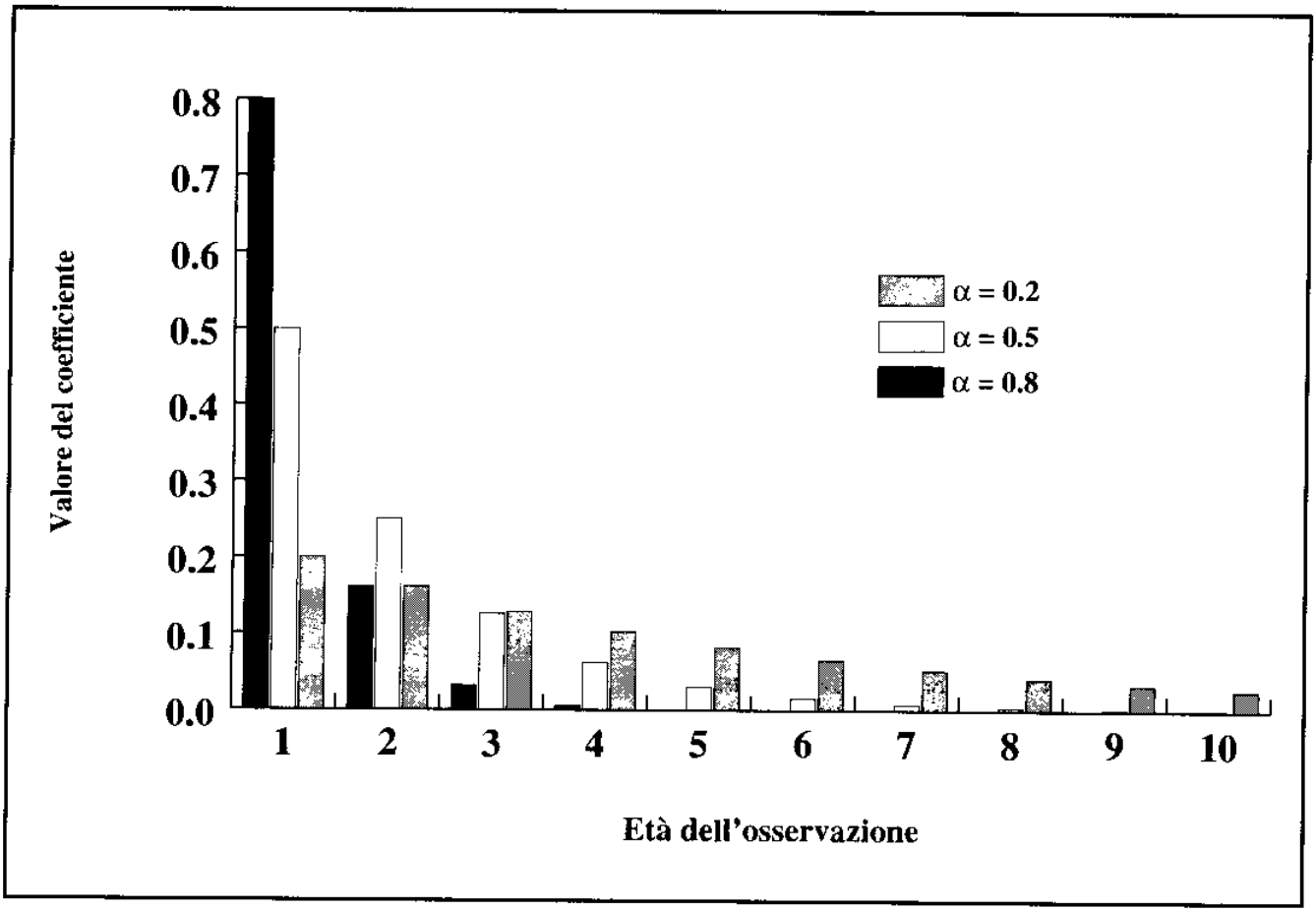


Figura 9.8 *Coefficienti per smorzare la media esponenziale*

bi i casi, si comincia con una stima di $S_1 = 0$, che dà priorità maggiore al nuovo processo. Bisogna notare che la media esponenziale segue i cambiamenti di comportamento del processo più velocemente della media semplice, e che il valore più grande di α provoca una reazione più rapida ai cambiamenti nel valore osservato.

Con SPN i processi più lunghi rischiano la starvation, se c'è un arrivo stabile di processi più brevi; d'altra parte, sebbene SPN riduca il vantaggio a favore dei processi più lunghi, non è ancora accettabile per un ambiente time sharing o orientato alle transazioni, a causa della mancanza del prerilascio. Guardando indietro all'analisi del caso peggiore descritto per FCFS, i processi A, B, C e D saranno ancora eseguiti nello stesso ordine, penalizzando pesantemente il processo C.

Shortest Remaining Time

La strategia dello Shortest Remaining Time (SRT: tempo rimanente più breve) è una versione di SPN con prerilascio: in questo caso, lo scheduler sceglie sempre il processo che ha il minor tempo di esecuzione rimanente previsto. Quando un nuovo processo è inserito nella coda dei processi Ready, può avere un tempo rimanente più breve di quello del processo in esecuzione al momento; di conseguenza, lo scheduler può dover prerilasciare ogni volta che un nuovo processo diventa Ready. Come per SPN, lo scheduler deve avere una stima del tempo di elaborazione per eseguire la funzione di selezione, e c'è il rischio di starvation per i processi più lunghi.

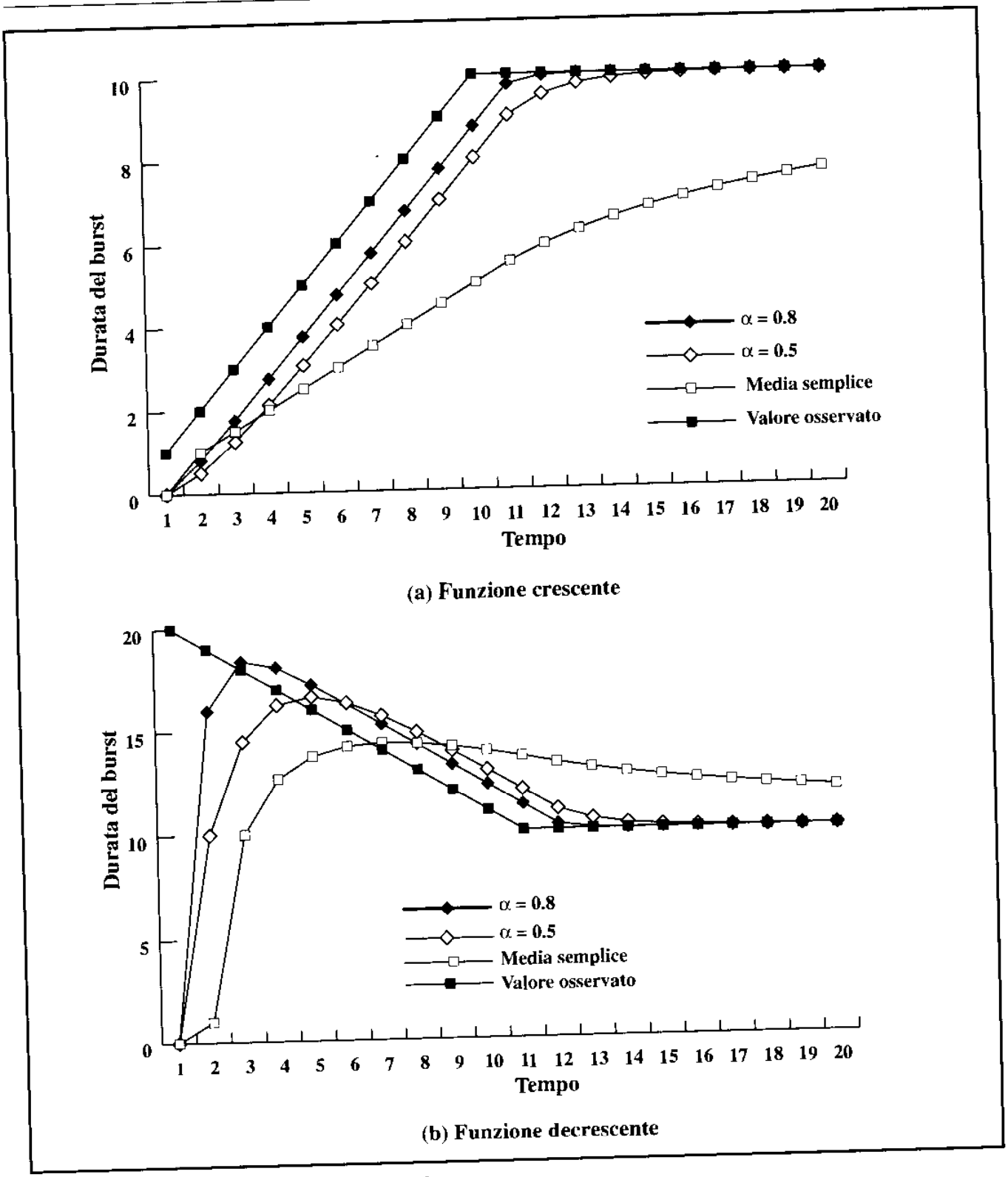


Figura 9.9 *Uso della media esponenziale*

SRT non avvantaggia i processi più lunghi, come fa FCFS, e a differenza del round robin, non si generano interrupt aggiuntivi, riducendo l'overhead. D'altra parte, i tempi di servizio trascorsi devono essere registrati, e questo contribuisce all'overhead. SRT potrebbe anche portare ad un tempo di turnaround superiore alle prestazioni di SPN, poiché si dà una preferenza immediata ai lavori brevi, a discapito di quelli lunghi in esecuzione.

Bisogna notare che nell'esempio precedente (Tabella 9.4) i tre processi più brevi sono subito serviti, ottenendo per ciascuno un tempo di turnaround normalizzato di 1.0.

Highest Response Ratio Next

Nella Tabella 9.4 si è usato il tempo di turnaround normalizzato che è il rapporto tra il tempo di turnaround e il tempo di servizio attuale, come valore discriminante. Si voglia minimizzare questo rapporto per ogni processo e il suo valor medio per tutti i processi. Sebbene questa sia una misura a posteriori, si può approssimarla con una misura a priori come criterio di selezione in uno scheduler senza prerilascio. Specificatamente, si consideri il seguente rapporto, in cui RR è il rapporto di risposta:

$$RR = \frac{w + s}{s}$$

dove

w = tempo speso ad aspettare il processore

s = tempo di servizio previsto

Se il processo con questo valore è allocato immediatamente, RR è uguale al tempo di turnaround normalizzato. Si noti che il valore minimo di RR è 1.0, che si ottiene quando un processo entra nel sistema per la prima volta.

Perciò, la regola di scheduling da seguire diventa: quando il processo in esecuzione termina o si blocca, si sceglie il processo Ready con il più grande valore di RR. Questo approccio si chiama Highest Response Ratio Next (HRRN: il più alto rapporto è il prossimo), ed è attraente perché tiene in considerazione l'età del processo: mentre sono favoriti i processi più brevi (un denominatore più piccolo conduce ad un maggior rapporto), invecchiare senza essere serviti aumenta il rapporto, così un processo più lungo supererà prima o poi i processi più brevi.

Come per SRT e SPN, per applicare HRRN si deve stimare il tempo di servizio previsto.

Feedback

Se non si hanno indicazioni sulla lunghezza relativa dei vari processi, allora non si può usare nessuno tra SPN, SRT e HRRN. Un altro modo per stabilire una preferenza per i job più brevi è penalizzare quei job che sono stati più a lungo in esecuzione. In altre parole, se non è possibile concentrarsi sul tempo d'esecuzione rimanente, ci si preoccupa del tempo speso in esecuzione fino ad ora.

Il modo per fare ciò è il seguente: lo scheduling è fatto su basi di prerilascio e si usa un meccanismo di priorità dinamico. Quando un processo entra per la prima volta, è posto nella coda Ready RQ0 (vedere Figura 9.4). Dopo la sua prima esecuzione, quando torna in stato Ready, è posto in RQ1. Ogni volta successiva, è spostato nella coda successiva con minore priorità: un processo più breve completerà prima la sua esecuzione, senza spostarsi molto in basso nella gerarchia delle code dei processi Ready, mentre un processo più lungo scenderà sempre più in basso. Perciò, i processi più brevi, arrivati da poco, sono favoriti rispetto a quelli

più vecchi e più lunghi. All'interno di ogni coda, eccetto nella coda a più bassa priorità, si usa un semplice meccanismo di FCFS. Una volta che un processo è nella coda con più bassa priorità, non può scendere più in basso, e si reinserisce ripetutamente in quest'ultima coda finché non ha completato la sua esecuzione; per questo motivo, questa coda è trattata con la strategia del round robin.

La Figura 9.10 illustra il meccanismo di scheduling con feedback, mostrando il cammino che un processo seguirà attraverso le varie code.[2] Questo approccio è conosciuto anche come feedback multilivello, nel senso che il sistema operativo alloca il processore ad un processo e, quando il processo si blocca od è prerilasciato, lo reinserisce in una delle diverse code di priorità.

Ci sono diverse variazioni a questo schema: una semplice versione suggerisce di eseguire il prerilascio nello stesso modo usato per il round robin: ad intervalli periodici. L'esempio riportato lo dimostra (Figura 9.5 e Tabella 9.4) per un quanto di tempo unitario: in questo caso il comportamento è simile a quello del round robin con un quanto di tempo unitario.

Il semplice schema appena esposto presenta però un problema: il tempo di turnaround dei processi più lunghi può aumentare in modo allarmante, ed è quindi possibile che si verifichi starvation se nuovi lavori entrano regolarmente nel sistema. Per compensare questo inconveniente, è possibile variare il tempo di prerilascio a seconda della coda: un processo schedulato

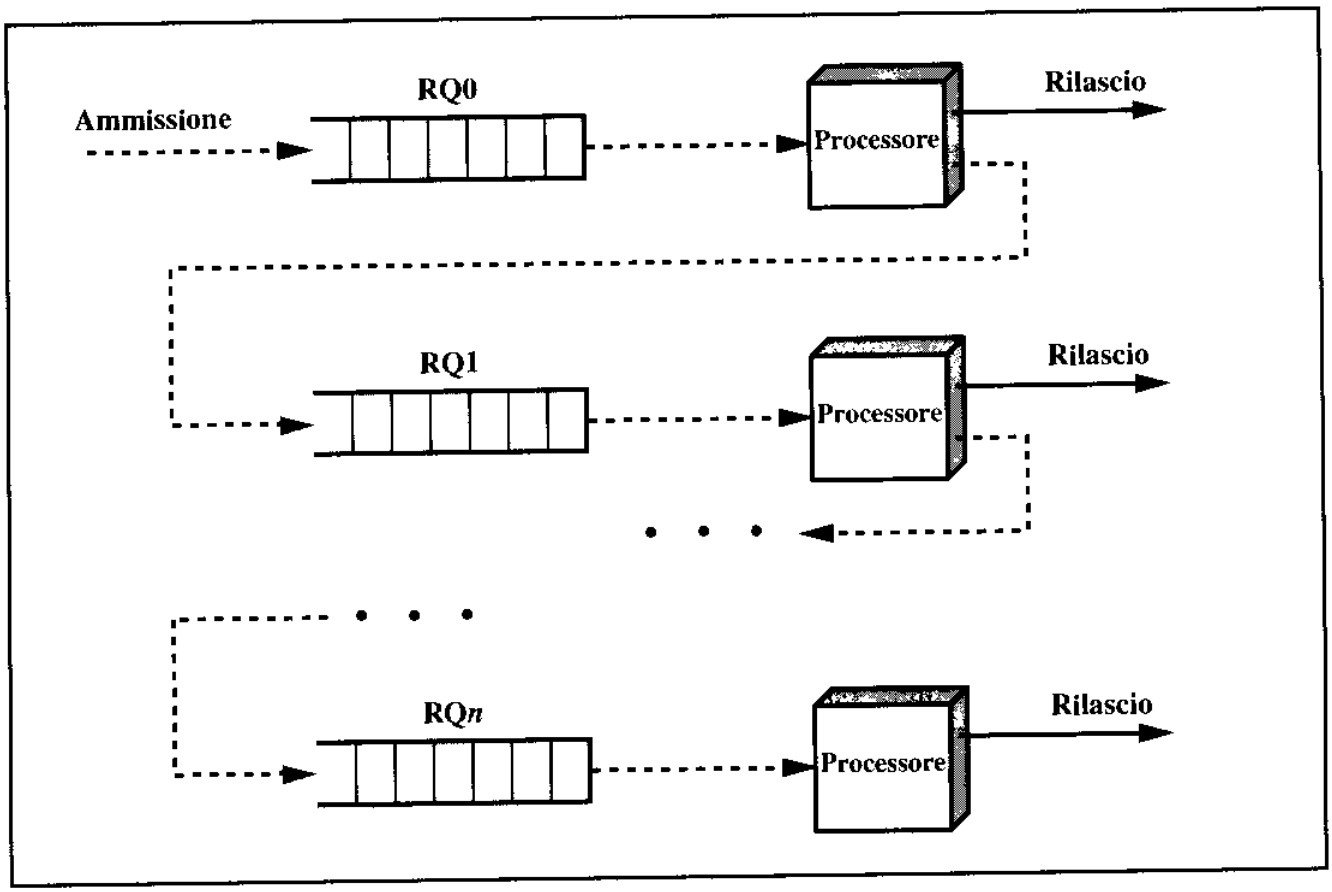


Figura 9.10 *Scheduling con feedback*

[2] Le linee punteggiate sono usate per evidenziare che questo è un diagramma di una sequenza di tempo piuttosto che una descrizione statica delle possibili transizioni, come nella Figura 9.4.

da RQ0 è autorizzato all'esecuzione per un'unità di tempo e poi è prerilasciato; un processo schedulato da RQ1 è autorizzato all'esecuzione per due unità di tempo, e così via. In generale, un processo schedulato da RQ_i è autorizzato all'esecuzione per 2^i unità di tempo prima del prerilascio: l'applicazione di questo schema al nostro esempio è illustrata in Figura 9.5 e nella Tabella 9.4.

Anche con la concessione di una maggiore quantità di tempo alla priorità più bassa, un processo più lungo può ancora soffrire di starvation. Un possibile rimedio consiste nella promozione di un processo ad una coda con priorità più alta, dopo che ha aspettato per un certo periodo il servizio nella coda in cui si trova.

Confronto sulle prestazioni

Chiaramente, le prestazioni delle varie strategie di scheduling rappresentano un fattore critico nella scelta della strategia di scheduling. È anche impossibile fare un confronto completo perché le prestazioni relative dipenderanno da svariati fattori come la distribuzione di probabilità dei tempi di servizio dei processi, l'efficienza dello scheduling e del meccanismo di cambio di processo, la natura delle richieste di I/O e le prestazioni del sottosistema di I/O. In ogni caso, si tenterà di trarre di seguito alcune conclusioni generali.

Analisi delle code

In questa sezione, si farà uso delle formule basilari relative alle code, con le comuni ipotesi di arrivo secondo la distribuzione Poissoniana e i tempi di servizio esponenziali. Un riassunto di questi concetti si può trovare nell'Appendice A.

Innanzitutto, bisogna osservare che qualunque strategia, che scelga il prossimo elemento da servire indipendentemente dal tempo di servizio, obbedisce alla seguente relazione:

$$\frac{T_q}{T_s} = \frac{1}{1-\rho}$$

dove

T_q = tempo di turnaround; tempo totale trascorso nel sistema, aspettando ed in esecuzione

T_s = tempo di servizio medio; tempo medio trascorso in stato Running

ρ = utilizzazione del processore

In particolare, uno scheduler basato sulla priorità, nel quale la priorità di ogni processo è assegnata indipendentemente dal tempo di servizio previsto, fornisce lo stesso tempo di turnaround medio e lo stesso tempo di turnaround normalizzato medio della semplice strategia FCFS. Inoltre la presenza o l'assenza di prerilascio non fa differenza su queste medie.

Ad eccezione del round robin e di FCFS, le varie strategie di scheduling considerate finora selezionano i processi in base al tempo di servizio previsto. Sfortunatamente, risulta abbastanza difficile sviluppare modelli analitici chiusi per queste strategie, comunque, ci si può fare un'idea delle prestazioni relative di tali algoritmi di scheduling, a confronto con FCFS, considerando uno scheduling a priorità in cui la priorità è basata sul tempo di servizio.

Se lo scheduling è fatto in base alla priorità, e se i processi sono assegnati ad una categoria di priorità in base al tempo di servizio, emergeranno delle differenze. La Tabella 9.5 mostra le formule che si ottengono quando si suppongono due classi di priorità, con diversi tempi di servizio per ogni classe. Nella tabella, λ si riferisce al tasso di arrivo. Questi risultati possono essere generalizzati a qualunque numero di classi di priorità (per esempio, vedere [MART72] per un riassunto di queste formule). Si noti che le formule differiscono a seconda dello scheduling usato: senza prerilascio o con prerilascio. Nel caso di scheduling con prerilascio, si suppone che un processo con più bassa priorità sia interrotto non appena un processo con più alta priorità diventa Ready.

Ad esempio, si consideri il caso di due classi di priorità, con un ugual numero di processi in ingresso in ogni classe e con tempo di servizio medio, della classe a più bassa priorità, cinque volte maggiore di quello della classe a priorità più alta. Perciò, si desidera dare la precedenza ai processi più brevi. La Figura 9.11 mostra il risultato complessivo: dando la preferenza ai job più brevi, si migliora il tempo di turnaround medio normalizzato e, come ci si può aspettare, il miglioramento è maggiore se si usa il prerilascio. Si noti comunque che le prestazioni complessive non ne sono molto influenzate.

Emergono differenze significative quando si considerano le due classi di priorità separatamente. La Figura 9.12 mostra i risultati per la priorità più alta, cioè per i processi più

Tabella 9.5 *Formule per code a server singolo con due categorie di priorità*

Ipotesi:

1. Tasso di arrivo secondo la distribuzione Poissoniana
2. Gli elementi con priorità 1 sono serviti prima degli elementi con priorità 2
3. Allocazione First-in, first-out dei processi con uguale priorità
4. Nessun elemento è interrotto mentre è servito
5. Nessun elemento lascia la coda (le chiamate perse sono ritardate)

(a) Formule generali

$$\lambda = \lambda_1 + \lambda_2$$

$$\rho_1 = \lambda_1 T_{s1}; \quad \rho_2 = \lambda_2 T_{s2}$$

$$\rho = \rho_1 + \rho_2$$

$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$

$$T_q = \frac{\lambda_1}{\lambda} T_{q1} + \frac{\lambda_2}{\lambda} T_{q2}$$

(b) Nessuna interruzione: tempo di servizio esponenziale

$$T_{q1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$$

$$T_{q2} = T_{s2} + \frac{T_{q1} + T_{s1}}{1 - \rho}$$

(c) Disciplina di code Preemptive-resume; tempo di servizio esponenziale

$$T_{q1} = 1 + \frac{\rho_1 T_{s1}}{1 - \rho_1}$$

$$T_{q2} = 1 + \frac{1}{1 - \rho}\left(\rho_1 T_{s1} + \frac{\rho T_s}{1 - \rho}\right)$$

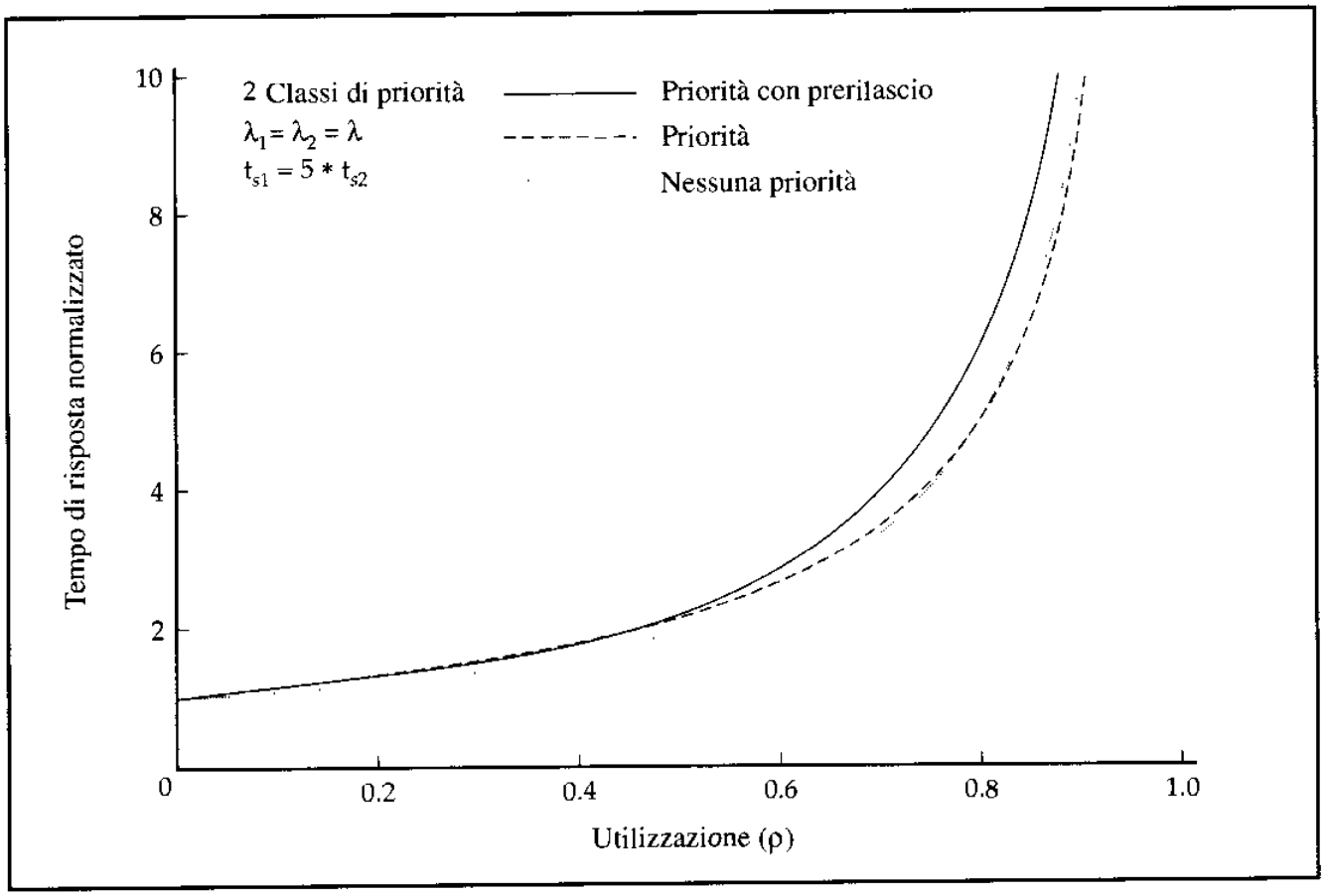


Figura 9.11 *Tempo di risposta totale normalizzato*

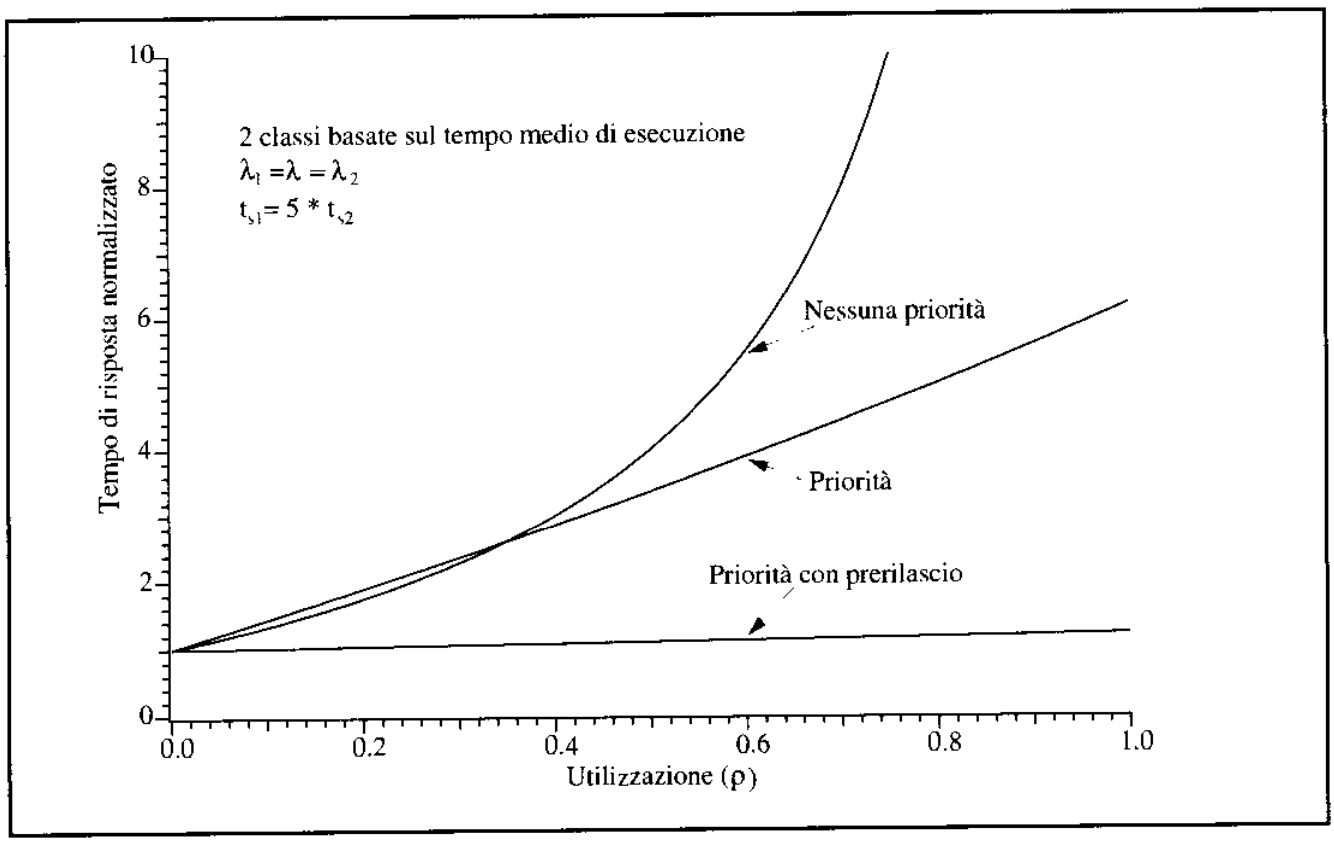


Figura 9.12 *Tempi di riposta normalizzati per i processi più brevi*

brevi. Per fare un confronto, la linea superiore nel grafico suppone che le priorità non siano usate, ma che si stiano considerando semplicemente le prestazioni relative a quella metà dei processi che hanno più breve tempo di esecuzione; le altre due linee suppongono che a questi processi siano assegnate più alte priorità. Quando il sistema è avviato usando lo scheduling a priorità senza prerilascio, i miglioramenti sono significativi, anche se con l'uso del prerilascio i miglioramenti sono maggiori.

La Figura 9.13 mostra la stessa analisi per i processi più lunghi, a priorità minori: come ci si aspettava, tali processi vedono le loro prestazioni degradare con questo tipo di scheduling.

Modello di simulazione

Alcune difficoltà della modellazione analitica sono superate usando la simulazione ad eventi discreti, che consente la modellazione di una buona varietà di strategie. Lo svantaggio della simulazione consiste nel fatto che i risultati per un dato "run" sono applicabili soltanto per quella raccolta particolare di processi e per quel particolare insieme di ipotesi: comunque, può sempre servire a ricavare utili intuizioni.

I risultati di uno di tali studi sino riportati in [FINK88]. La simulazione coinvolgeva 50000 processi con intervallo di arrivo di $\lambda = 0.8$ e un tempo di servizio medio $T_s = 1$. Perciò, si suppone che l'utilizzazione del processore sia $\rho = \lambda T_s = 0.8$ e si noti che si sta misurando solamente un punto di utilizzazione.

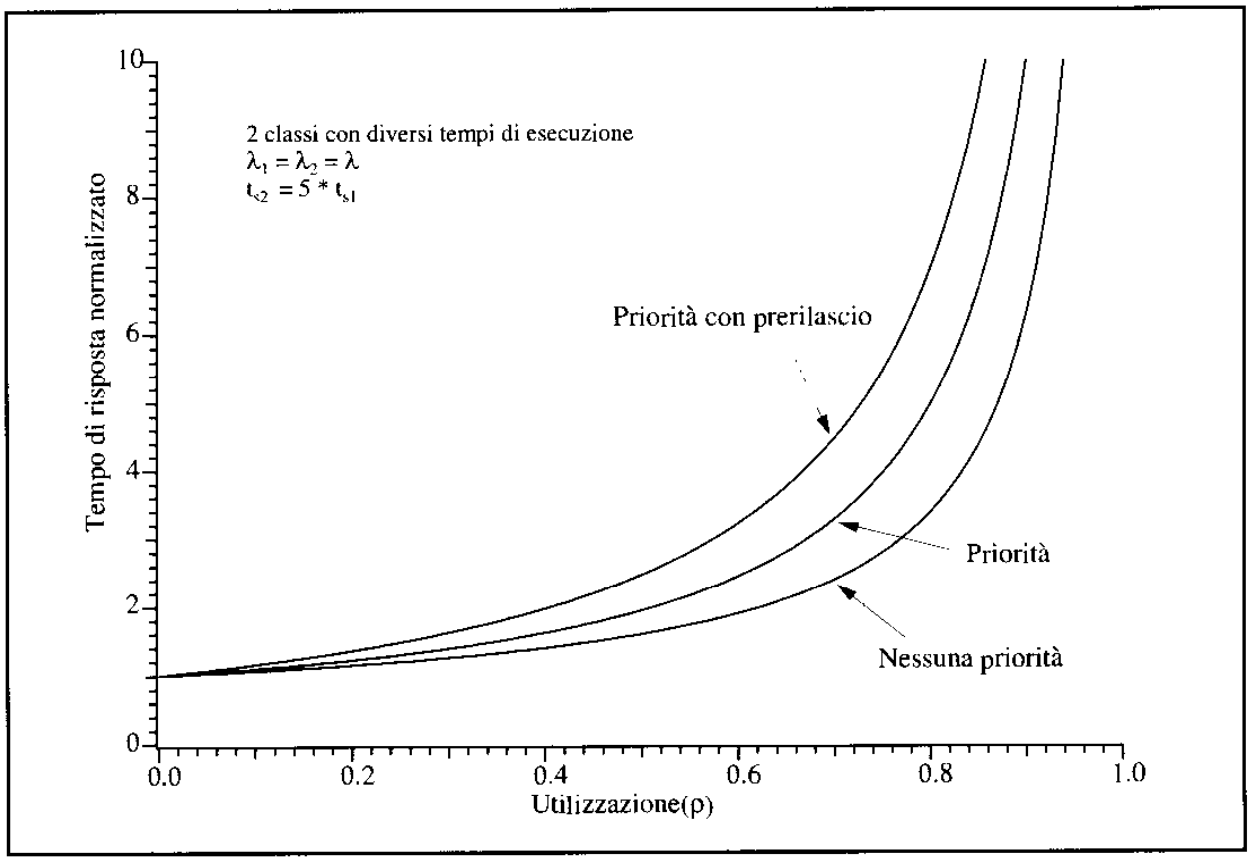


Figura 9.13 *Tempi di riposta normalizzati per i processi più lunghi*

Per presentare i risultati, i processi sono raggruppati in percentili del tempo di servizio, ognuno dei quali contiene 500 processi: i 500 processi con il più breve tempo di servizio sono nel primo percentile; eliminati questi, i 500, tra i rimanenti, con i più brevi tempi di servizio sono nel secondo percentile, e così via. Questo consente di vedere l'effetto delle varie strategie applicate ai processi come una funzione della lunghezza del processo.

La Figura 9.14 mostra il tempo di turnaround normalizzato e la Figura 9.15 mostra il tempo medio di attesa. Dal tempo di turnaround si può vedere che le prestazioni di FCFS sono molto sfavorevoli, poiché un terzo dei processi ha un tempo di turnaround dieci volte più grande del tempo di servizio; inoltre questi sono i processi più brevi. D'altra parte, il tempo assoluto di attesa è uniforme, come ci si aspetta, poiché lo scheduling è indipendente dal tempo di servizio. Le figure mostrano le prestazioni del round robin, usando un quanto di tempo unitario. Esclusi i processi più brevi, che sono eseguiti in meno di un quanto, il RR conduce ad un tempo di turnaround normalizzato di circa 5 per tutti i processi, trattandoli equamente. Lo shortest-process-next ha migliori prestazioni del round robin, eccetto che per i processi più brevi. Lo shortest-remaining-time, la versione di SPN con prerilascio, ha prestazioni migliori di SPN escluso quel 7% di processi più lunghi. Quindi, tra le strategie senza prerilascio, FCFS favorisce i processi lunghi, mentre SPN favorisce quelli brevi. L'highest-response-ratio-next è un compromesso tra questi due effetti, ed è infatti confermato nelle figure. Infine la figura mostra lo scheduling con feedback (FB) con un quanto uniforme e fisso per ogni coda di priorità. Come ci si aspettava, FB ha buone prestazioni per i processi brevi.

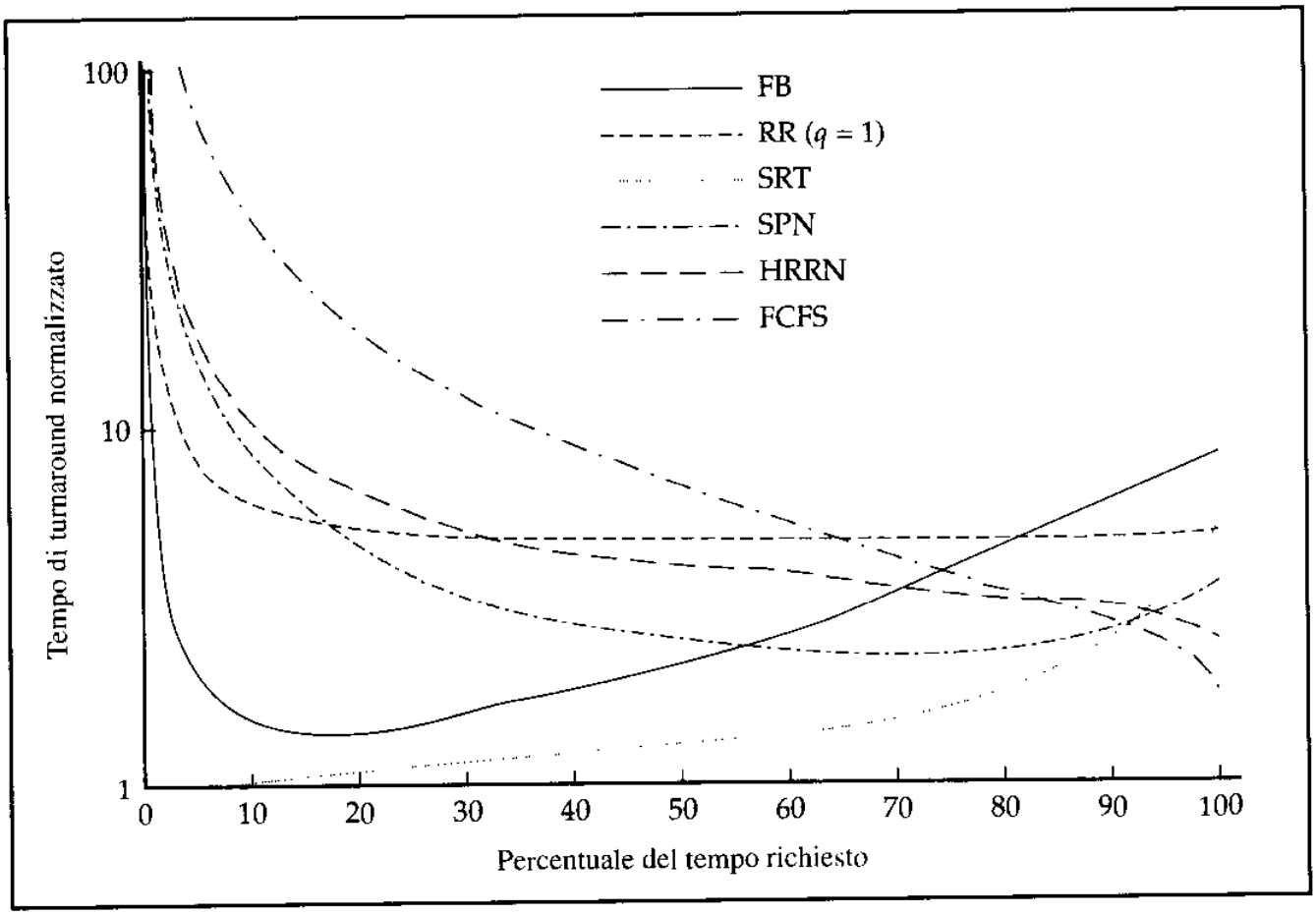


Figura 9.14 *Risultati della simulazione per il turnaround time normalizzato*

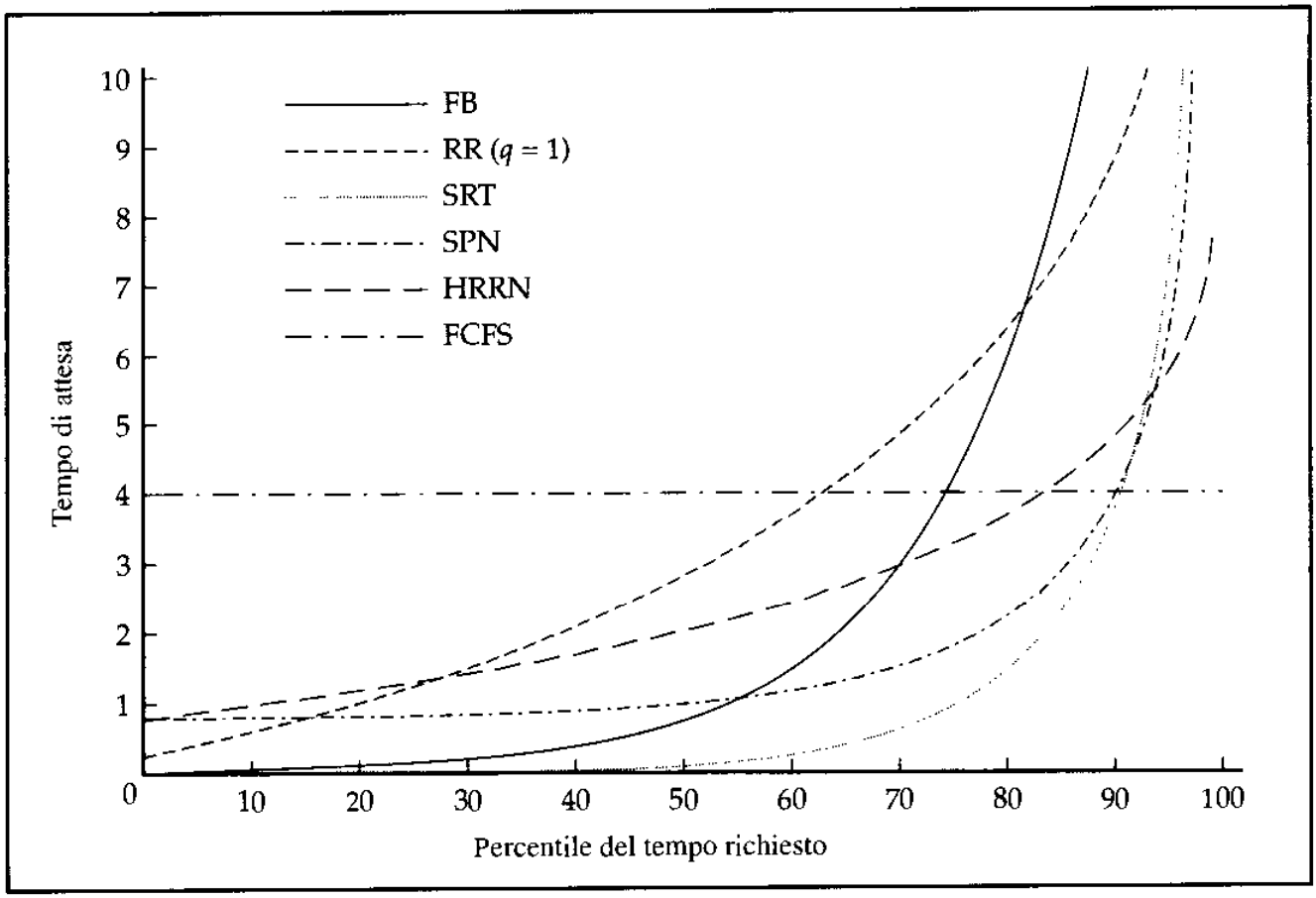


Figura 9.15 *Risultati della simulazione per il tempo di attesa*

Fair-share scheduling

Tutti gli algoritmi di scheduling fin qui discussi trattano un insieme di processi come una singola collezione di processi, da cui selezionare il prossimo processo da eseguire: può essere suddivisa dalla priorità, altrimenti è omogeneo.

In un sistema multiutente, se i job o le applicazioni dell'utente singolo possono essere organizzati come molti processi (o thread), allora la collezione di processi ha una struttura, che non è riconosciuta dallo scheduler tradizionale. Dal punto di vista dell'utente, la preoccupazione non sono le prestazioni di un processo, ma piuttosto come sono le prestazioni dell'insieme di processi, che costituiscono la singola applicazione. Perciò, sarebbe interessante decidere la strategia di scheduling da usare sulla base dei gruppi dei processi. Questo approccio è conosciuto come fair-share scheduling (scheduling con suddivisione equa). Il concetto può essere esteso a gruppi di utenti, anche se ogni utente è rappresentato da un singolo processo. Per esempio, nei sistemi time-sharing, si possono considerare tutti gli utenti di un dato dipartimento come membri dello stesso gruppo. Le decisioni di scheduling possono allora essere prese in modo da assicurare ad ogni gruppo di processi un servizio simile. Perciò, se un gran numero di persone di un dipartimento accede al sistema, si vorrebbe vedere che il peggioramento del tempo di risposta colpisca soprattutto le persone di quel dipartimento piuttosto che gli utenti degli altri dipartimenti.

Il termine **fair-share** indica la filosofia che sta dietro a tale scheduler. Ad ogni utente è assegnato un peso di qualche tipo che definisce la suddivisione delle risorse di sistema di quel-

l'utente come una parte dell'uso totale di quelle risorse. In particolare, ad ogni utente si assegna una parte del processore. Tale schema opera approssimativamente in modo lineare, così se l'utente A ha due volte il peso dell'utente B, allora alla lunga, l'utente A dovrebbe poter fare il doppio del lavoro dell'utente B. L'obiettivo di questo scheduler è controllare l'uso per dare meno risorse agli utenti che hanno avuto più della loro giusta parte e più risorse a quelli che ne hanno avuto di meno.

Sono state fatte diverse proposte per fair-share scheduler [HENR84, KAY88, LARM75, WOOD86]: in questa sezione, si descriverà lo schema proposto in [HENR84] e implementato su un certo numero di sistemi UNIX. Lo schema è conosciuto come fair-share scheduler (FSS). Esso considera la storia dell'esecuzione di un gruppo di processi, insieme con la storia dell'esecuzione di ogni singolo processo per prendere la decisione di scheduling. Il sistema divide la comunità degli utenti in un insieme di gruppi con suddivisioni giuste (fair-share group), ed alloca una parte delle risorse del processore per ogni gruppo. Perciò, se ci fossero quattro gruppi, ognuno avrebbe il 25% dell'uso del processore. In effetti, ogni fair-share group è dotato di una macchina virtuale che lavora più lentamente dell'intero sistema, in proporzione.

Lo scheduling è fatto in base alla priorità, che tiene conto della priorità del processo, il suo recente uso del processore e il recente uso del processore da parte del gruppo al quale il processo appartiene. Più alto è il valore numerico della priorità, minore è la priorità. Le seguenti formule si applicano al processo j nel gruppo k:

$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{2} + \frac{\text{GCPU}_k(i-1)}{4 \times W_k}$$

$$\text{CPU}_j(i) = \frac{U_j(i-1)}{2} + \frac{\text{CPU}_j(i-1)}{2}$$

$$\text{GCPU}_k(i) = \frac{\text{GU}_k(i-1)}{2} + \frac{\text{GCPU}_k(i-1)}{2}$$

dove

$P_j(i)$ = priorità del processo j all'inizio dell'intervallo i

Base_j = priorità base del processo j

$U_j(i)$ = utilizzazione del processore da parte del processo j nell'intervallo i

$\text{GU}_k(i)$ = utilizzazione totale del processore da parte dei processi del gruppo k nell'intervallo i

$\text{CPU}_j(i)$ = utilizzazione del processore mediato sui pesi esponenziali, da parte del processo j nell'intervallo i

$\text{GCPU}_k(i)$ = utilizzazione del processore mediato sui pesi esponenziali del gruppo k nell'intervallo i

W_k = peso assegnato al gruppo k, con il vincolo per cui $0 \le W_k \le 1$ e $\sum_k W_k = 1$

A ogni processo è assegnata una priorità di base. Questa priorità diminuisce non appena il processo, oppure il gruppo cui il processo appartiene, usa il processore. In entrambi i casi, si tiene conto della media di utilizzazione del processore tramite media esponenziale, con $\alpha = 0.5$. Nel caso di utilizzazione da parte del gruppo, la media è normalizzata dividendola per il peso di quel gruppo. Maggiore è il peso assegnato al gruppo, minore è l'influenza dell'utilizzazione sulla priorità.

La Figura 9.16 mostra un esempio in cui il processo A è in un gruppo e il processo B e il processo C sono in un secondo gruppo: ogni gruppo ha un peso di 0.5. Si supponga che tutti i processi siano processor bound e solitamente pronti per l'esecuzione. Tutti i processi hanno una priorità base di 60. L'utilizzazione del processore è così misurata: il processore è interrotto 60 volte al secondo, e durante ogni interruzione il campo che tiene traccia dell'uso del processore, nel processo in esecuzione, è incrementato, come pure il campo del gruppo corrispondente. Una volta al secondo si ricalcolano le priorità.

Tempo	Processo A			Processo B			Processo C		
	Priorità	Processore	Gruppo	Priorità	Prucessore	Gruppo	Priorità	Processore	Gruppo
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		•	•						
		•	•						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					•	•			•
					•	•			•
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		•	•						
		•	•						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						•		•	•
						•		•	•
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		•	•						
		•	•						
		78	78						
5	98	39	39	70	3	18	76	15	18

Figura 9.16 *Esempio di Fair Share Scheduler - tre processi, due gruppi [BACH86]*

In figura, il processo A è schedulato per primo; dopo un secondo, è prerilasciato, ed a questo punto i processi B e C hanno la priorità più alta, e si schedula il processo B. Alla fine della seconda unità di tempo, il processo A ha la più alta priorità: la sequenza di schedulazione si ripete, perché il kernel schedula i processi nell'ordine: A, B, A, C, A, B, ecc. Perciò il 50% del processore è allocato per il processo A, che costituisce un gruppo, e l'altro 50% è allocato per B e C, che costituiscono un altro gruppo.

9.3 Lo scheduling tradizionale di UNIX

In questa sezione si esaminerà lo scheduler tradizionale di UNIX, che è usato sia in SVR3 che in 4.3 BSD UNIX. Questi sistemi sono principalmente mirati ad ambienti interattivi time-sharing. L'algoritmo di scheduling è progettato per fornire un buon tempo di risposta agli utenti interattivi, assicurando che i job eseguiti in background a bassa priorità non soffrano di starvation. Sebbene questo algoritmo sia stato rimpiazzato nei moderni sistemi UNIX, vale la pena di esaminarlo perché è rappresentativo degli algoritmi di scheduling per time sharing. Lo schema di scheduling per SVR4 comprende facilitazioni per i requisiti in tempo reale, e quindi la sua trattazione è rimandata al capitolo 10.

Lo scheduler tradizionale di UNIX utilizza il feedback multilivello, sfruttando la strategia del round robin all'interno di ciascuna coda di priorità. Il sistema fa uso del prerilascio ogni secondo, cioè, se un processo in esecuzione non si blocca o non termina in un secondo, è prerilasciato. La priorità è basata sul tipo di processo e la storia dell'esecuzione. Si applicano le seguenti formule:

$$P_j(i) = \text{Base}_j + \frac{\text{CPU}j(i-1)}{2} + \text{nice}_j$$

$$\text{CPU}_j(i) = \frac{U_j(\text{i})}{2} + \frac{\text{CPU}_j(i-1)}{2}$$

dove

$P_j(i)$ = priorità del processo j all'inizio dell'intervallo i: valori più bassi equivalgono a più alte priorità

Base_j = priorità base del processo j

$U_j(i)$ = utilizzazione del processore da parte del processo j nell'intervallo i

$\text{CPU}_j(i)$ = utilizzazione media del processore pesata esponenzialmente, da parte del processo j nell'intervallo i

nice_j = fattore di assestamento, controllabile dall'utente

La priorità di ogni processo è ricalcolata una volta al secondo, nel momento in cui si prende una nuova decisione di scheduling. Lo scopo della priorità di base è dividere tutti i processi in bande fissate di livelli di priorità. I componenti CPU e nice si limitano ad evitare che un processo esca dalla banda cui è stato assegnato (assegnato dal livello di priorità di base). Queste bande

sono utilizzate per ottimizzare l'accesso ai dispositivi a blocchi (come i dischi) e permettere al sistema operativo di rispondere rapidamente alle chiamate di sistema. In ordine decrescente di priorità, le bande sono le seguenti:

- Swapper
- Controllo dei dispositivi di I/O a blocchi
- Manipolazione di file
- Controllo dei dispositivi di I/O a caratteri
- Processi utente.

Questa gerarchia dovrebbe dare il più efficiente uso dei dispositivi di I/O. All'interno della banda dei processi utente, l'uso della storia di esecuzione tende a penalizzare i processi processor

	Processo A		Processo B		Processo C	
Tempo	Priorità	Contatore di CPU	Priorità	Contatore di CPU	Priorità	Contatore di CPU
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Figura 9.17 *Esempio dello scheduling tradizionale dei processi in UNIX [BACH86]*

bound, alle spese dei processi I/O bound: anche questo migliorerebbe l'efficienza. Accoppiato con lo schema del round robin con prerilascio, la strategia di scheduling è ben organizzata per soddisfare i requisiti del time sharing generale.

Un esempio di scheduling di processi è mostrato in Figura 9.17. I processi A, B e C sono creati nello stesso momento con priorità base di 60 (si ignorerà il valore nice). Il clock interrompe il sistema 60 volte al secondo ed aumenta il contatore del processo in esecuzione. L'esempio suppone che nessuno dei processi si blocchi e che nessun altro processo sia pronto per l'esecuzione. Confrontare questa figura con la 9.16.

9.4 Sommario

Il sistema operativo deve prendere tre tipi di decisioni di scheduling rispetto all'esecuzione dei processi. Lo scheduling a lungo termine determina quando sono ammessi nuovi processi nel sistema, lo scheduling a medio termine è una parte della funzione di trasferimento su disco dei processi, e determina quando un programma va caricato totalmente o parzialmente in memoria in modo da eseguirlo; infine lo scheduling a breve termine determina quale processo Ready sarà eseguito successivamente dal processore. Questo capitolo si concentra sugli argomenti relativi allo scheduling a breve termine.

Una varietà di criteri è usata nella progettazione dello scheduling a breve termine: alcuni di questi criteri sono in relazione al comportamento del sistema, come è percepito dall'utente individuale (orientati all'utente), mentre gli altri vedono l'efficacia totale del sistema andando incontro alle necessità di tutti gli utenti (orientati al sistema). Alcuni criteri sono in relazione specificatamente con misure quantitative delle prestazioni, mentre altri sono più qualitativi. Dal punto di vista dell'utente, il tempo di risposta è generalmente la più importante caratteristica di un sistema, mentre dal punto di vista del sistema, è importante l'utilizzazione del processore, o il throughput.

Per prendere decisioni di scheduling a breve termine, fra tutti i processi pronti, sono stati sviluppati diversi algoritmi:

- **First come first served**: Seleziona il processo che ha il tempo di attesa più lungo.
- **Round robin**: Usa il time-slicing per limitare tutti i processi in fase di esecuzione ad un piccolo periodo d'uso del processore e ruota su tutti i processi Ready.
- **Shortest process next**: Seleziona il processo con il minore tempo di uso del processore previsto e non prerilascia il processo.
- **Shortest remaining time**: Seleziona il processo con il minore tempo di uso del processore previsto; un processo può essere prerilasciato quando un altro processo diventa Ready.
- **Highest response ratio next**: Basa la decisione di scheduling su una stima del tempo di turnaround normalizzato.
- **Feedback**: Crea un insieme di code per lo scheduling, ed alloca i processi in tali code a seconda della loro storia di esecuzione, o su altri criteri.

La scelta di un algoritmo per lo scheduling dipenderà dalle prestazioni previste e dalla complessità dell'implementazione.

9.5 Letture raccomandate

Generalmente tutti i testi sui sistemi operativi trattano l'argomento dello scheduling. Rigorose analisi delle code e varie politiche per lo scheduling sono presentate su [STUC85], [KLEI76] e [CONW67].

CONW67 Conway, R.; Maxwell, W. e Miller, L. *Theory of Scheduling*. Reading, MA: Addison-Wesley, 1967.

KLEI76 Kleinrock, L. *Queuing Systems, Volume II: Computer Applications*. New York: Wiley, 1976.

STUC85 Stuck, B. e Arthurs, E. *A Computer and Communication Network Performance Analysis Primer* Englewood Cliffs, NJ: Prentice Hall, 1985.

9.6 Problemi

9.1 Considerare il seguente insieme di processi

Nome processo	Tempo di arrivo	Tempo di esecuzione
1	0	3
2	1	5
3	3	2
4	9	5
5	12	5

Si faccia la stessa analisi di questo insieme come descritto nella Tabella 9.4 e in Figura 9.5

9.2 Ripetere il Problema 9.1 con il seguente insieme:

Nome processo	Tempo di arrivo	Tempo di esecuzione
A	0	1
B	1	9
C	2	1
D	3	9

9.3 Provare che, fra gli algoritmi di scheduling senza prerilascio, SPN permette di ottenere il minimo tempo di attesa media, supponendo che tutti i job arrivino contemporaneamente.

9.4 Si consideri la seguente sequenza di tempi di burst per un processo: 6, 4, 6, 4, 13, 13, 13, e si supponga che il valore iniziale stimato sia 10. Produrre un grafico simile a quelli di Figura 9.9

9.5 Si consideri la seguente coppia di equazioni come un alternativa all'equazione (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n$$

$$X = \min[\text{Ubound}, \max[\text{Lbound}, (\beta S_{n+1})]]$$

Dove Ubound e Lbound sono il limite superiore e quello inferiore sul valore stimato di T precedentemente scelti. Il valore X è usato nell'algoritmo shortest process next invece del valore S_{n+1}. Quali funzioni sviluppano α e β e quali sono gli effetti di loro valori maggiori o minori?

9.6 In un sistema uniprocessore senza prerilascio la coda dei processi Ready contiene 3 job al tempo t immediatamente dopo il completamento di un job. Questi job hanno tempi arrivo t_1, t_2, t_3, e tempi stimati di esecuzione r_1, r_2, r_3, rispettivamente.
La Figura 9.18 mostra la crescita lineare della loro risposta media nel tempo. Si usi quest'esempio per trovare una variante a HRRN, conosciuta come **minimax response ratio** scheduling (minimo dei rapporti massimi di risposta), che minimizza la risposta media massima per un dato gruppo di job, ignorando arrivi successivi (*Suggerimento*: si decida subito quale job schedulare per ultimo).

9.7 Si provi che l'algoritmo di minimax response ratio trattato nel precedente problema minimizza la risposta media massima per un dato gruppo di job (*Suggerimento*: focalizzare l'attenzione sul job che totalizzerà il più grande rapporto di risposta e su tutti i job

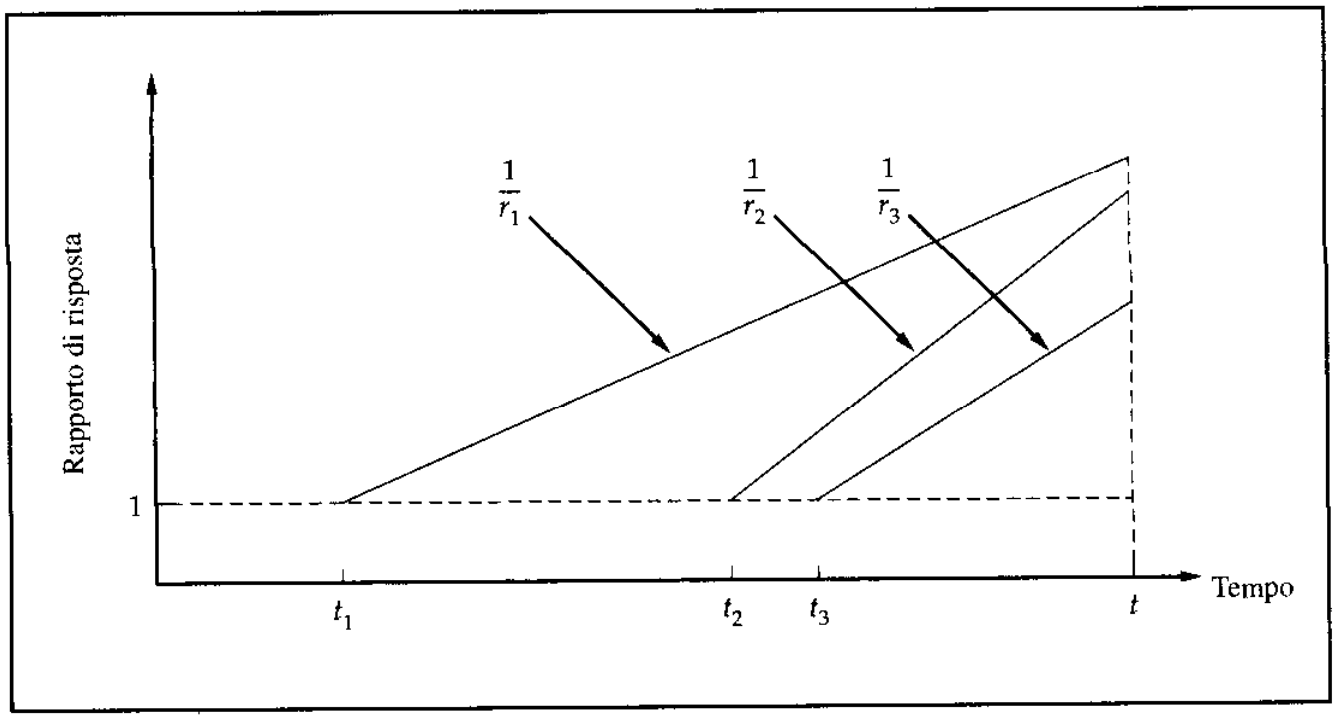


Figura 9.18 *Tempo di risposta in funzione del tempo*

eseguiti prima di esso. Si consideri lo stesso sottoinsieme di job schedulati in un qualsiasi altro ordine, e si osservi il rapporto di risposta del job eseguito per ultimo tra di essi. Si noti che questo sottoinsieme può essere adesso mischiato con altri job presi dall'insieme totale)

9.8 Si definisca il tempo di risposta R come il tempo medio totale che un processo passa aspettando un servizio. Si mostri che per la politica FIFO, $R = S/(1-\rho)$, dove S è il tempo di servizio medio.

9.9 Un processore è multiprogrammato ad una velocità infinita fra tutti i processi presenti nella coda Ready senza overhead. (Questo è un modello ideale di uno scheduling round-robin sui processi Ready usando dei quanti di tempo molto piccoli rispetto al tempo di esecuzione). Dimostrare che con un input di Poisson da una sorgente infinita con tempi di servizio esponenziali, la media del tempo di risposta Rx di un processo con tempo di servizio x è data da $Rx= x/(1-\rho)$. (*Suggerimento*: si riguardino le equazioni delle code nell'Appendice A, quindi si consideri q, la dimensione media delle code nel sistema, all'arrivo del processo preso in considerazione)

9.10 La maggior parte degli scheduler round-robin usano una dimensione fissata del quanto. Si cerchino argomentazioni a favore di un piccolo quanto e viceversa, a favore un grande quanto. Si paragonino e si contrappongano i tipi di sistema e di job sui quali le argomentazioni sono state applicate. Ce ne sono certi per cui *entrambe* sono ragionevoli?

9.11 In un sistema a code i nuovi job devono attendere un po' prima di essere serviti. Mentre un job attende, la sua priorità cresce linearmente nel tempo da 0 a velocità α. Un processo attende fintanto che la sua priorità non arriva alla priorità del processo in servizio; in quel momento, quindi, incomincia a condividere il processore in modo equo con gli altri processi in esecuzione usando lo scheduling round-robin, mentre la sua priorità continua a crescere, ma con velocità minore β. L'algoritmo è detto selfish round robin perché i job in esecuzione cercano (invano) di monopolizzare il processore aumentando la loro priorità in continuazione. Si usi la Figura 9.19 per mostrare che il tempo di risposta medio R_x per un processo al tempo di servizio x è dato da:

$$R_x = \frac{s}{1-\rho} + \frac{x-s}{1-\rho'}$$

Dove

$$\rho = \lambda s \qquad \rho' = \rho\,(1 - \frac{\beta}{\alpha}) \qquad 0 \le \beta \le \alpha$$

Supponendo che il tempo di arrivo e quello di servizio siano distribuiti esponenzialmente con media $1/\lambda$ e s rispettivamente. (*Suggerimento*: Si consideri il sistema totale e i 2 sottosistemi separatamente).

9.12 Un sistema interattivo che usa lo scheduling round-robin e lo swapping cerca di dare una risposta garantita alle richieste irrisorie come segue: dopo aver completato un ciclo round-robin su tutti i processi Ready, il sistema determina quanto tempo concedere ad ogni

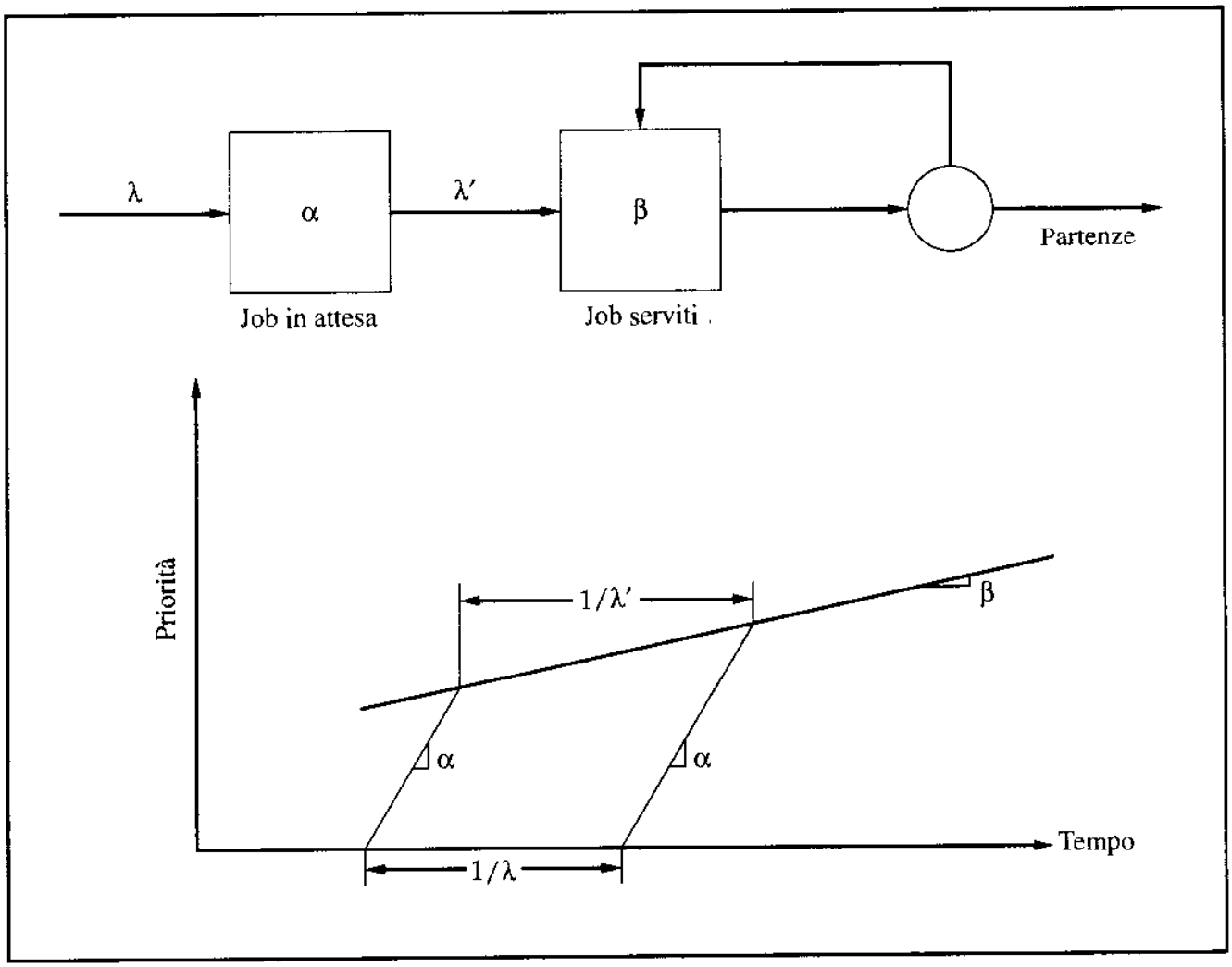


Figura 9.19 *Selfish Round Robin*

processo Ready durante il ciclo successivo, dividendo il tempo di risposta massimo per il numero dei processi che richiedono il servizio. Questa strategia è praticabile?

9.13 Che tipo di processo è generalmente favorito da uno scheduler con code con feedback multilivello, un processo processor-bound o uno I/O bound? Si dia una piccola spiegazione.

9.14 In uno scheduling basato sulla priorità, lo scheduler passa il controllo ad un particolare processo solo se non ci sono processi con una priorità più alta in stato Ready. Si supponga che non vi siano altre informazioni per fare lo scheduling dei processi, e che le priorità dei processi, stabilite al tempo di creazione dei processi, non cambino mai. In un sistema che lavora in tale modo perché è pericoloso utilizzare la soluzione di Dekker per il problema della mutua esclusione? Lo si spieghi dicendo *quale* evento indesiderato potrebbe presentarsi e *come*.

9.15 Cinque job batch, da A fino a E, arrivano praticamente allo stesso tempo all'unità di elaborazione. Essi hanno un tempo di esecuzione stimato di 15, 9, 3, 6 e 12 minuti rispettivamente; le loro priorità (definite esternamente) sono rispettivamente 4, 7, 3, 1 e 6, dove

10 è la priorità massima. Per ognuno dei seguenti algoritmi di scheduling si determini il tempo di turnaround di ciascun processo e il turnaround medio di tutti i job. Non si tenga conto dell'overhead dovuto al cambio di processo e si spieghi come si è arrivati alla risposte. Negli ultimi tre casi, si supponga di eseguire un solo processo per volta e di finire solo quando tutti i job sono completamente processor-bound.

a. Round robin con un quanto che vale 1

b. Scheduling con priorità

c. FCFS (si esegue in ordine 15, 9, 3, 6 e 12)

d. Shortest job first.

Appendice 9A Tempo di risposta

Il tempo di risposta è il tempo che un sistema richiede per reagire ad un certo input. In una transazione interattiva, può essere definito come il tempo che intercorre da quando l'ultimo tasto della tastiera è stato premuto dall'utente, a quando il risultato è visualizzato dal computer sullo schermo. Per diversi tipi di applicazione, serve una definizione leggermente differente; ma in generale è il tempo che il sistema si prende per rispondere alla richiesta di eseguire un compito particolare.

Idealmente, piacerebbe a tutti che il tempo di risposta fosse il più breve possibile per ogni applicazione. Tuttavia spesso un tempo di risposta più breve richiede un costo maggiore: il costo, infatti, dipende da due fattori:

- **Potenza di elaborazione**: Più veloce è il computer, più breve sarà il tempo di risposta; ovviamente, aumentare la potenza del processore significa aumentare i costi.
- **Competizione tra richieste**: Dare tempi di risposta rapidi a certi processi può penalizzarne altri.

Per questi motivi si deve valutare il livello del tempo di risposta rispetto al costo di ottenere quel tempo di risposta.

La Tabella 9.6, basata su [MART88], elenca sei intervalli generali di tempi di risposta. Si incontrano difficoltà di progettazione quando è richiesto un tempo di risposta inferiore ad un secondo: una tale richiesta è tipica di un sistema che controlla o interagisce con un'attività esterna, come in una catena di montaggio: questa richiesta è immediata. Quando consideriamo un'interazione uomo-macchina, come in un'applicazione di inserimento dati, siamo in un intervallo di risposta di tipo conversazionale. In questo caso c'è ancora una richiesta per un tempo di risposta breve, ma una durata accettabile potrebbe essere difficile da valutare.

Il tempo di risposta rapido è la base per avere produttività nelle applicazioni interattive, cosa confermata da più studi [SHNE84; THAD81; GUYN88]. Questi studi dimostrano che se un computer ed un utente interagiscono di pari passo, in modo che nessuno dei due debba attendere l'altro, la produttività cresce significativamente, il costo del lavoro fatto al computer perciò si

Tabella 9.6 *Livelli del tempo di risposta*

Maggiore di 15 secondi

Impossibile avere interazioni con domanda e risposta. Per certi tipi di applicazioni, gli utenti potrebbero essere contenti di sedersi davanti ad un terminale ed aspettare più di 15 secondi per avere una risposta ad una loro semplice domanda. Tuttavia per persone molto impegnate un'attesa di 15 secondi non è tollerabile. Se vi è un tale ritardo, il sistema dovrebbe essere progettato in modo da permettere all'utente di passare ad altre attività e rimandare la risposta ad un altro momento.

Maggiore di 4 secondi

Questo tempo è troppo lungo per una conversazione che richiede all'operatore di mantenere delle informazioni in memoria a breve termine (quella dell'operatore, non del computer!). Tali ritardi possono limitare fortemente attività di soluzione di problemi, e sono frustranti nell'inserimento dei dati. Tuttavia dopo una importante chiusura, ritardi da 4 a 15 secondi possono essere tollerati.

Da 2 a 4 secondi

Un ritardo sopra i 2 secondi può essere proibitivo per operazioni al terminale che richiedono un alto livello di concentrazione. Un'attesa da 2 a 4 secondi al terminale può sembrare estremamente lunga quando un utente è personalmente teso a completare il più presto possibile quello che sta facendo. Di nuovo un ritardo in questo livello può essere accettabile dopo che interviene una più piccola chiusura.

Meno di 2 secondi

Quando l'utente al terminale deve ricordare molte risposte, il tempo di risposta deve essere breve. Più dettagli si devono ricordare, maggiore è il bisogno di un tempo di risposta inferiore ai 2 secondi. Per attività al terminale complicate, 2 secondi rappresentano un importante tempo di risposta limite.

Tempi di risposta sotto il secondo

Certi lavori che richiedono un'attenzione intensa, specialmente con applicazioni grafiche, richiedono un tempo di risposta molto breve per mantenere l'interesse dell'utente e l'attenzione per un lungo periodo.

Tempi di risposta in decimi di secondo

In risposta alla pressione di un tasto della tastiera o del mouse, vedere il carattere sullo schermo o selezionare un oggetto sullo schermo deve essere quasi istantaneo - meno di 0,1 secondi dopo l'azione. Un'interazione con il mouse richiede una risposta estremamente veloce, se il progettista vuole evitare di usare una sintassi aliena (con comandi, punteggiatura mnemonica).

abbassa e la qualità tende ad aumentare. Era un dato di fatto largamente accettato che una risposta relativamente lenta, fino a 2 secondi, fosse accettabile per molte applicazioni interattive poiché una persona doveva pensare al task successivo. Adesso, tuttavia, sembra che la produttività aumenti se si ottengono veloci tempi di risposta.

I risultati riportati sul tempo di risposta sono basati su un'analisi delle transazioni in linea. Una transazione si compone di un comando utente dal terminale e della risposta del sistema, e rappresenta l'unita di lavoro fondamentale per gli utenti in linea. Può essere divisa in due sequenze temporali:

- **Tempo di risposta dell'utente**: È il tempo che intercorre da quando l'utente riceve una risposta ad un comando a quando invia un altro comando. Si dice anche *think time* (tempo per pensare).
- **Tempo di risposta del sistema**: È il tempo che intercorre da quando l'utente invia un comando a quando la risposta è visualizzata sul terminale.

Come esempio sull'effetto di una riduzione del tempo di risposta, la Figura 9.20 mostra i risultati di uno studio effettuato da ingegneri usando un programma di grafica CAD per il progetto di chip di circuiti integrati e schede [SMIT83]: ogni transazione consiste di un comando inviato da un ingegnere, che modifica l'immagine a schermo. Il risultato mostra che la frequenza delle transazioni aumenta se il tempo di risposta scende, ed aumenta drasticamente se il tempo di risposta scende sotto 1 secondo. Quello che sta succedendo è che, quando il tempo di risposta del sistema scende, così fa anche il tempo di risposta dell'utente: tutto questo è in relazione con gli effetti della memoria a breve termine e con l'attenzione.

Un'altra area dove il tempo di risposta è diventato critico è l'uso del World Wide Web, sia su Internet sia in un'intranet. Il tempo che impiega una normale pagina web ad essere visualizzata può cambiare notevolmente. Il tempo di risposta può essere stimato in base al livello di coinvolgimento dell'utente; in particolare alcuni sistemi con un tempo di risposta molto veloce

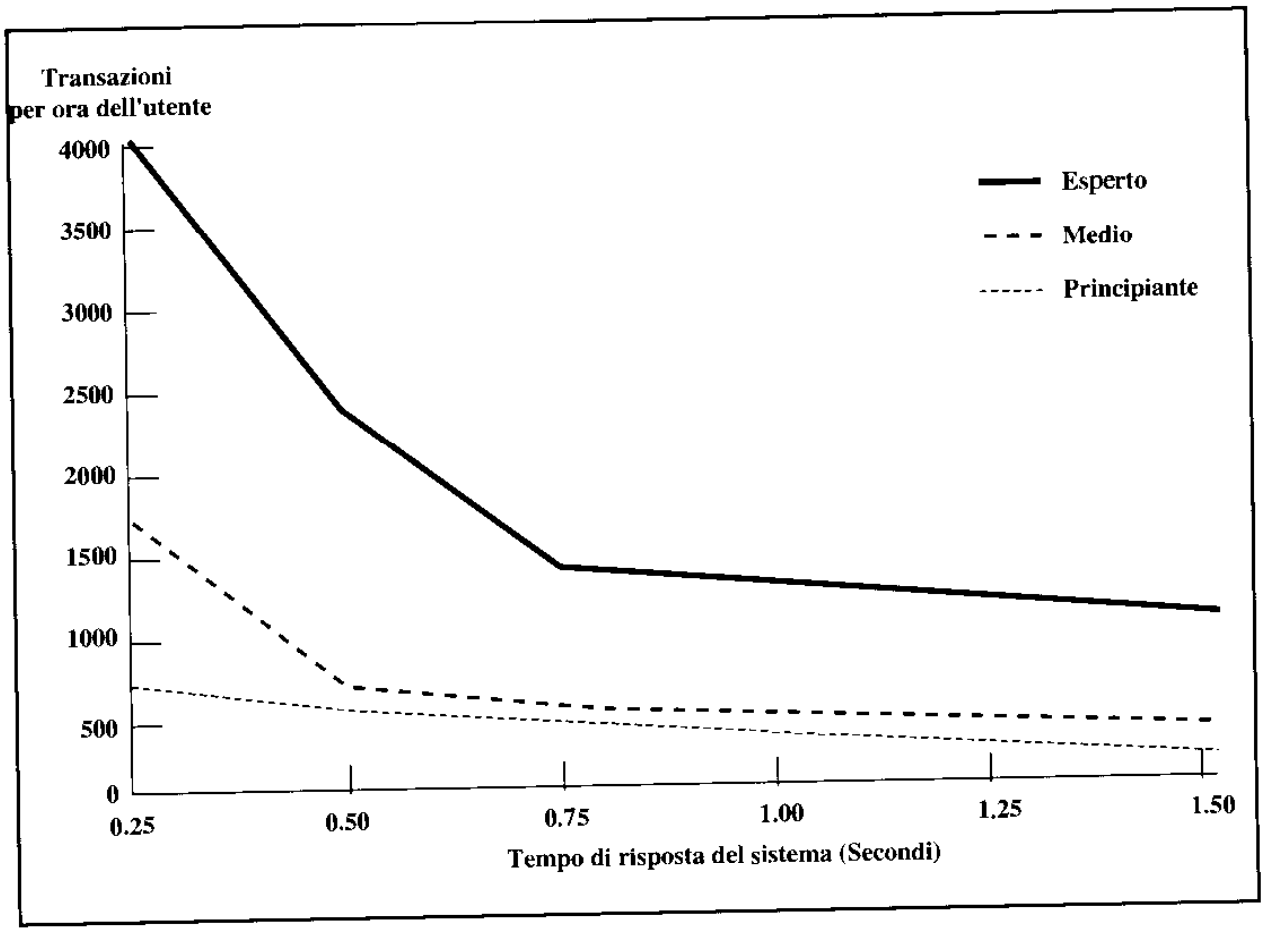


Figura 9.20 *Tempi di risposta per una funzione grafica*

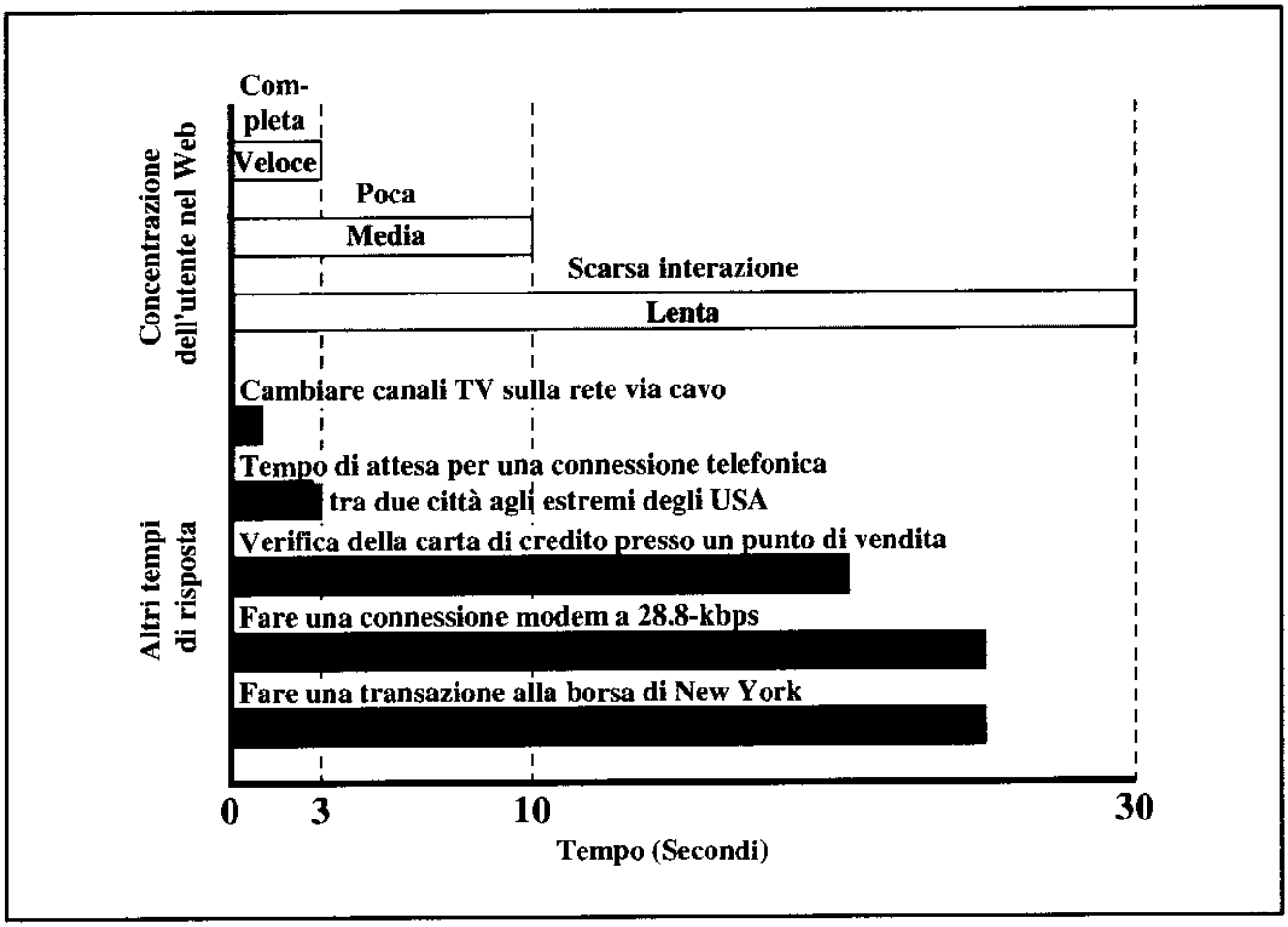


Figura 9.21 *Tempi di risposta richiesti [SEVC96]*

tendono a richiedere maggiore attenzione da parte dell'utente. Come ci mostra la Figura 9.21, sistemi Web con un tempo di risposta di 3 secondi o migliore mantengono un alto livello di attenzione dell'utente. Con un tempo di risposta tra i 3 e i 10 secondi si perde una parte della concentrazione dell'utente, e i tempi di risposta superiori ai 10 secondi scoraggiano l'utente, che può semplicemente terminare la sessione.

CAPITOLO 10

SCHEDULING MULTIPROCESSORE E IN TEMPO REALE

Questo capitolo continua lo studio dello scheduling di processi. Si inizia con un esame delle questioni sorte dalla disponibilità di più di un processore, esplorando alcuni problemi di progettazione. In seguito si dà uno sguardo allo scheduling di processi su un sistema multiprocessore; si esaminano poi considerazioni di progettazione piuttosto diverse per lo scheduling multiprocessore di thread. La seconda sezione di questo capitolo tratta lo scheduling in tempo reale, iniziando con una discussione sulle caratteristiche dei processi in tempo reale e poi esamina la natura del processo di scheduling. Sono esaminati due approcci allo scheduling in tempo reale, lo scheduling con scadenze e lo scheduling a frequenza monotona.

10.1 Scheduling multiprocessore

Quando un sistema di elaborazione contiene più di un processore, sorgono alcuni problemi nuovi nella fase di progettazione della funzione di scheduling. Si comincia con una breve panoramica sui multiprocessori e poi si fanno considerazioni abbastanza diverse per lo scheduling a livello di processo ed a livello di thread.

Possiamo classificare i sistemi multiprocessore come segue:

- **Multiprocessore accoppiato lascamente (loosely coupled), o cluster:** si compone di un insieme di sistemi relativamente autonomi, in cui ogni processore possiede la sua propria memoria principale e canali di I/O. Parleremo di questo tipo di configurazione nel Capitolo 14.

- **Processori specializzati funzionalmente:** ad esempio, un processore di I/O. In questo caso, c'è un processore master di tipo generale; i processori specializzati sono controllati dal processore master cui forniscono servizi. I problemi relativi ai processori di I/O vengono discussi nel Capitolo 11.
- **Multiprocessore accoppiato strettamente(tightly coupled):** si compone di un insieme di processori, che condividono una memoria principale comune e sono sotto il controllo integrato di un sistema operativo.

Questa sezione si rivolge all'ultima categoria e più specificatamente verso i problemi relativi allo scheduling.

Granularità

Un buon metodo per caratterizzare i multiprocessori e poterli confrontare con le altre architetture è considerare la granularità della sincronizzazione, o grado di sincronizzazione, tra i processi in un sistema. È possibile distinguere cinque categorie di parallelismo, che differiscono nel grado di granularità. Queste sono riassunte nella Tabella 10.1, che è stata adattata da [GEHR87] e [WOOD89].

Parallelismo indipendente

Con il parallelismo indipendente, non c'è sincronizzazione esplicita tra i processi: ciascuno rappresenta un'applicazione o un job separato, indipendente. Questo tipo di parallelismo è solitamente usato in un sistema time-sharing: ogni utente esegue una particolare applicazione, come ad esempio l'elaboratore di testi oppure il foglio elettronico. Il multiprocessore fornisce lo stesso servizio di un monoprocessore multiprogrammato, ma essendo disponibili più processori, il tempo di risposta medio agli utenti sarà minore.

Tabella 10.1 *Granularità della sincronizzazione e processi*

Misura della granularità	Descrizione	Intervallo di sincronizzazione (Istruzioni)
Fine	Parallelismo interno ad un solo flusso di istruzioni	<20
Media	Elaborazione parallela o multitasking entro una singola applicazione	20-200
Grossolana	Multiprocessing di processi concorrenti in un ambiente di multiprogrammazione	200-2000
Molto grossolana	Elaborazione distribuita attraverso nodi di rete, per formare un singolo ambiente di calcolo	2000-1M
Indipendente	Processi multipli non collegati	Non disponibile

È possibile raggiungere un simile guadagno di prestazioni fornendo ad ogni utente un personal computer od una workstation; se devono essere condivisi file o informazioni, allora i sistemi individuali devono essere collegati insieme in un sistema distribuito supportato da una rete. Quest'approccio è esaminato nel Capitolo 14. D'altra parte, un singolo sistema multiprocessore condiviso in molti casi è più economico di un sistema distribuito, permettendo economie di scala nei dischi e in altre periferiche.

Parallelismo a grana grossa e molto grossa

Con il parallelismo a grana grossa e molto grossa, c'è una sincronizzazione tra i processi, ma ad un livello, appunto, a grana molto grossa. Questo genere di situazione viene facilmente trattata come un insieme di processi concorrenti eseguiti su un monoprocessore multiprogrammato e può essere supportata su un multiprocessore, con nessuno o pochi cambiamenti al software dell'utente.

Un semplice esempio di un'applicazione in grado di sfruttare l'esistenza di un multiprocessore è data in [WOOD89]. Gli autori hanno sviluppato un programma che prende una descrizione dettagliata dei file che hanno bisogno della ricompilazione per ricostruire un pezzo di software e determina quali di queste compilazioni (di solito tutte) possono essere eseguite simultaneamente. Il programma genera poi un processo per ogni compilazione parallela. Gli autori riferiscono che l'aumento di velocità su un multiprocessore supera realmente quello che ci si sarebbe aspettati sommando semplicemente il numero dei processori in uso, per le sinergie nelle cache del buffer del disco (argomento trattato nel Capitolo 11) e per la condivisione del codice del compilatore, che è caricato in memoria una volta sola.

In generale, ogni insieme di processi concorrenti, che hanno bisogno di comunicare o sincronizzarsi, può beneficiare dell'uso di un'architettura a multiprocessore. Nel caso di interazioni tra processi poco frequenti, un sistema distribuito può fornire un buon supporto. Tuttavia, se l'interazione è più frequente, allora l'overhead di comunicazione attraverso la rete può negare parte dell'aumento di velocità potenziale. In quel caso, l'organizzazione a multiprocessore fornisce il supporto più efficace.

Parallelismo a granularità media

Nel Capitolo 4 si è visto che una singola applicazione può essere efficacemente implementata come un insieme di thread entro un singolo processo. In questo caso, il parallelismo potenziale di un'applicazione deve essere specificato esplicitamente dal programmatore. Tipicamente, sarà richiesto un grado di coordinamento e d'interazione tra i thread dell'applicazione piuttosto alto, che porta ad un livello di sincronizzazione di granularità media. Mentre il parallelismo a granularità indipendente, molto grossolana e grossolana può essere supportato sia da un monoprocessore multiprogrammato sia da un multiprocessore, con poco o nessun impatto sulla funzione di scheduling, bisogna riesaminare lo scheduling quando si tratta di scheduling di thread. Le decisioni di scheduling che riguardano un thread possono influenzare le prestazioni dell'intera applicazione, poiché i vari thread di un'applicazione interagiscono molto frequentemente. Si tornerà su quest'argomento più avanti nel corso di questa stessa sezione.

Parallelismo a granularità fine

Il parallelismo a granularità fine rappresenta un uso molto più complesso del parallelismo rispetto a quello trovato nell'uso dei thread. Sebbene sia stato fatto molto lavoro sulle applicazioni altamente parallele, questa è, fino ad ora, un'area specializzata e frammentata, con molti approcci diversi. Un buon quadro generale è [ALMA98].

Problemi di progettazione

Lo scheduling su un multiprocessore riguarda tre argomenti correlati:

- L'assegnazione dei processi ai processori
- L'uso della multiprogrammazione su singoli processori
- L'allocazione effettiva di un processo.

Osservando questi tre argomenti, è importante ricordare che l'approccio scelto dipenderà, in generale, dal grado di granularità delle applicazioni e dal numero di processori disponibili.

Assegnazione dei processi ai processori

Se si suppone che l'architettura del multiprocessore sia uniforme, nel senso che nessun processore possiede un particolare vantaggio fisico rispetto all'accesso alla memoria principale o ai dispositivi di I/O, allora il più semplice approccio di scheduling consiste nel trattare i processori come un gruppo di risorse, ed assegnare i processi ai processori su richiesta. Sorge allora l'interrogativo sulla scelta dell'assegnazione: statica o dinamica?

Se un processo è assegnato permanentemente ad un processore, dalla sua attivazione al suo completamento, allora ogni processore mantiene una coda a breve termine dedicata. Un vantaggio di quest'approccio è che ci può essere meno overhead nella funzione di scheduling, perché l'assegnazione del processore è fatta una volta sola e per sempre. Inoltre, l'uso di processori dedicati permette una strategia nota come scheduling di gruppo o gang-scheduling, come discusso più tardi.

Uno svantaggio dell'assegnazione statica è che un processore può essere in ozio, con una coda vuota, mentre un altro processore ha un carico arretrato. Per evitare questa situazione, si può usare una coda comune: tutti i processi finiscono in una coda globale e sono schedulati su un qualsiasi processore disponibile; quindi durante la sua vita, un processo può essere eseguito su processori diversi in tempi diversi. In un'architettura a memoria condivisa strettamente accoppiata, le informazioni sul contesto dei processi saranno disponibili a tutti i processori, perciò il costo dello scheduling di un processo sarà indipendente dall'identità del processore su cui è schedulato.

In entrambi i casi ora descritti, è necessario un modo per assegnare i processi ai processori. Si possono usare due approcci: master/slave e alla pari (peer). Con un'architettura master/slave, le funzioni chiave del kernel del sistema operativo sono sempre eseguite su un processore particolare: gli altri processori possono solo eseguire programmi utente. Il master è responsabile

dello scheduling dei job: quando il processo è attivo, se lo slave ha bisogno di un servizio (ad es. una chiamata di I/O), deve mandare una richiesta al master ed aspettare che sia effettuato il servizio; quest'approccio è abbastanza semplice e richiede piccoli miglioramenti ad un sistema operativo monoprocessore in multiprogrammazione. La risoluzione dei conflitti è semplificata perché un solo processore ha il controllo di tutta la memoria e delle risorse di I/O, ma ci sono due svantaggi in quest'approccio: (1) un fallimento del master distrugge l'intero sistema, e (2) il master può diventare un collo di bottiglia per le prestazioni.

In un architettura di tipo peer, il sistema operativo può essere eseguito su qualsiasi processore, ed ogni processore auto-effettua lo scheduling dell'insieme dei processi disponibili. Quest'approccio complica il sistema operativo: esso deve assicurarsi che due processori non scelgano lo stesso processo e che i processi non siano in qualche modo persi dalla coda. Si devono impiegare tecniche specifiche per risolvere e sincronizzare richieste concorrenti alle risorse.

C'è, naturalmente, una gamma di approcci tra questi due estremi. Si può fornire un sottoinsieme di processori dedicati all'elaborazione del kernel, invece che uno solo; un altro approccio consiste semplicemente nel gestire la differenza tra le necessità dei processi di kernel e quelle degli altri processi mediante la priorità e la storia delle esecuzioni.

Uso di multiprogrammazione su singoli processori

Quando ogni processo è assegnato staticamente ad un processore per tutta la durata della sua esecuzione, sorge una nuova questione: quel processore dovrebbe essere multiprogrammato? La prima reazione del lettore potrebbe essere di meravigliarsi di questa domanda; sembra uno spreco vincolare il processore ad un singolo processo, che può essere bloccato di frequente, in attesa di I/O o per problemi di concorrenza/sincronizzazione.

Nel multiprocessore tradizionale, che si occupa di processi paralleli a granularità grossa o a sincronizzazione indipendente (vedere Tabella 10.1), è chiaro che ogni singolo processore dovrebbe essere in grado di alternarsi tra numerosi processi per raggiungere un'alta utilizzazione, e perciò migliori prestazioni. Tuttavia, per applicazioni a granularità media eseguite su un multiprocessore con molti processori, la situazione è meno chiara: quando sono disponibili molti processori, non è più essenziale che ogni singolo processore sia occupato il più possibile, piuttosto, si è interessati a fornire, in media, la migliore prestazione alle applicazioni. Un'applicazione che si compone di un certo numero di thread può essere eseguita lentamente, a meno che tutti i suoi thread non siano disponibili ad essere eseguiti simultaneamente.

Allocazione del processo

L'ultimo argomento relativo allo scheduling multiprocessore è la selezione effettiva del processo da eseguire. Su un monoprocessore multiprogrammato, l'uso di priorità o di sofisticati algoritmi di scheduling basati sull'utilizzo passato, possono essere meglio di una strategia semplice come first-come-first-served (FCFS: il primo processo che arriva è servito). Quando si passa ai multiprocessori, queste complicazioni possono essere non necessarie o addirittura controproducenti, e un approccio più semplice può essere più efficace, con minor overhead. Nel caso di scheduling di thread, entrano in gioco nuove questioni, che potrebbero essere più importanti della priorità o delle storie dell'esecuzione: esamineremo questi argomenti, uno alla volta.

Scheduling di processo

Nella maggior parte dei sistemi multiprocessore tradizionali, i processi non sono dedicati ai processori: c'è una coda singola per tutti i processori, o se si usa una sorta di schema a priorità qualsiasi, ci sono code multiple basate sulla priorità, che utilizzano tutte l'insieme comune dei processori. In ogni caso, possiamo vedere il sistema come un'architettura a coda multiserver.

Si consideri il caso di un sistema a due processori, ciascuno dei quali possiede la metà della velocità di elaborazione di un processore singolo. [SAUE81] riporta un'analisi di code, che confronta lo scheduling FCFS con round robin e shortest-remaining-time. Nel caso del round robin, si ipotizza che il quanto di tempo sia grande, rispetto all'overhead di cambio di processo, e piccolo in confronto con il tempo medio di servizio. I risultati dipendono criticamente dalla variabilità osservata nei tempi di servizio; una misura comune di variabilità è il coefficiente di variazione, C_s, definito da:

$$C_s = \frac{\sigma_s}{s}$$

dove

σ_s = deviazione standard del tempo di servizio

s = tempo di servizio medio

Un rapporto uguale a 1 corrisponderebbe ad un tempo di servizio esponenziale, tuttavia, le distribuzioni del tempo di servizio del processore di solito sono più variabili, e non sono inusuali valori di C_s uguali a 10, o anche maggiori.

La Figura 10.1a confronta il throughput di round-robin con il throughput di FCFS in funzione di C_s. Da notare che la differenza negli algoritmi di scheduling è molto ridotta nel caso del sistema a due processori. Con due processori, un processo con un tempo di servizio lungo nel caso FCFS è molto meno dirompente; gli altri processi possono utilizzare l'altro processore. Analoghi risultati sono mostrati in Figura 10.1b.

Lo studio in [SAUE81] ha ripetuto quest'analisi con un numero di ipotesi su grado di multiprogrammazione, misto di processi I/O bound e CPU bound ed uso di priorità: la conclusione generale è che la disciplina di scheduling specifica è molto meno importante con due processori rispetto ad uno solo. Dovrebbe essere evidente che questa conclusione è anche più forte all'aumentare del numero dei processori, quindi, una semplice disciplina FCFS o l'uso di FCFS entro uno schema a priorità statica può essere sufficiente per un sistema multiprocessore.

Scheduling di thread

Con i thread, il concetto di esecuzione è separato dal resto della definizione di un processo; un'applicazione può essere implementata come un insieme di thread, che cooperano e sono eseguiti nello stesso spazio di indirizzo concorrentemente.

Su un monoprocessore, i thread possono essere usati come un aiuto alla strutturazione dei programmi. e per sovrapporre l'I/O con l'elaborazione: a causa del costo minimo che un cambio di thread presenta, rispetto ad un cambio di processi, questi miglioramenti sono realizzati con poco costo. Tuttavia, la potenzialità dei thread diventa evidente in un sistema multiprocessore:

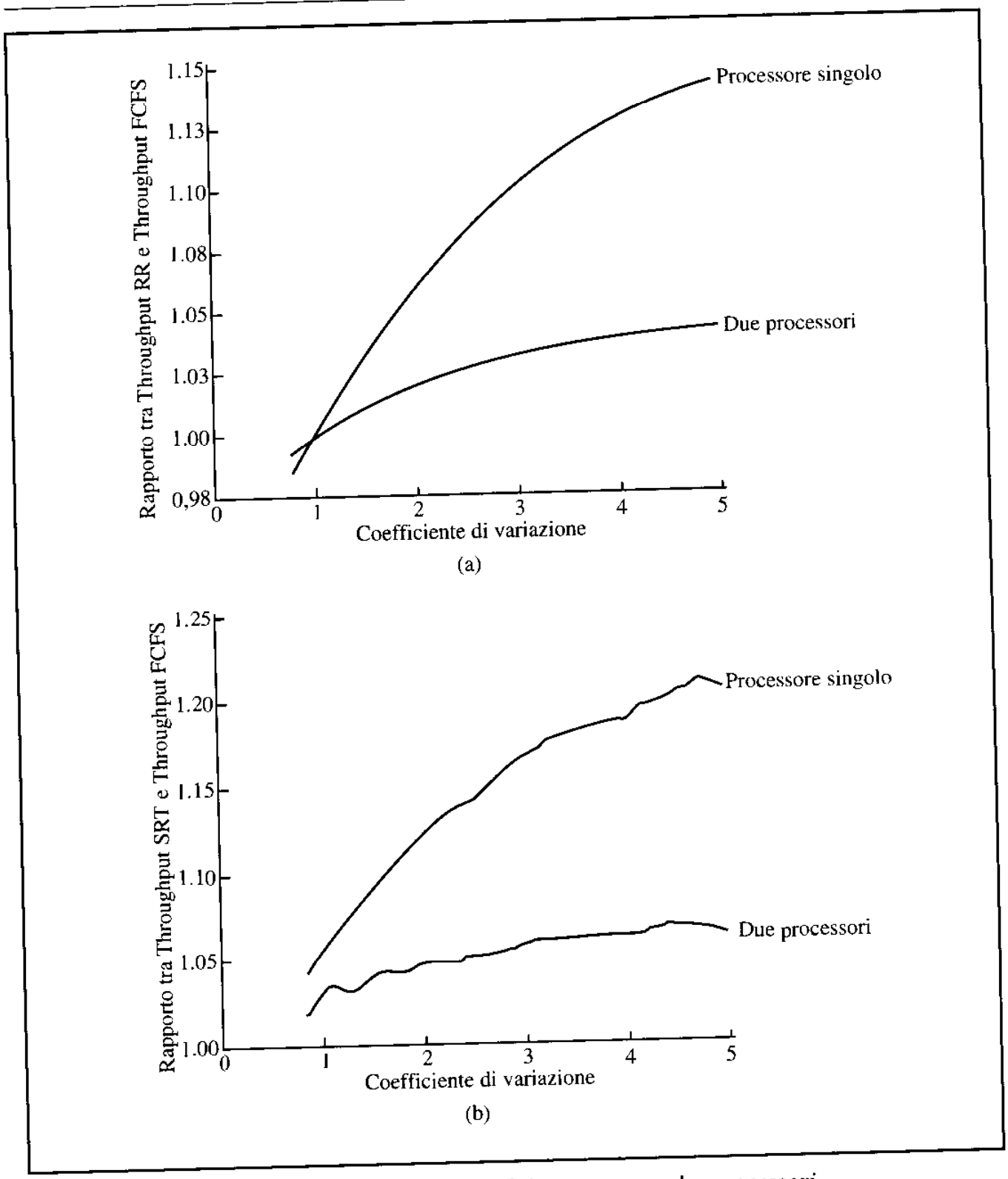


Figura 10.1 *Confronto delle prestazioni di scheduling per uno o due processori*

in quest'ambiente, i thread possono essere utilizzati per sfruttare il vero parallelismo in un'applicazione, e se i vari thread di un'applicazione sono eseguiti simultaneamente su processori separati, sono possibili guadagni evidenti nelle prestazioni. Tuttavia, si può dimostrare che per applicazioni che richiedono un'interazione significativa tra i thread (parallelismo a granularità media), piccole differenze nella gestione dei thread e nello scheduling possono avere un impatto

significativo sulle prestazioni [ANDE89]. L'area di scheduling dei thread su multiprocessori è un'area di ricerca attiva; la discussione fornisce qui una panoramica di argomenti chiave e di approcci.

Fra le molte proposte per lo scheduling multiprocessore di thread e l'assegnazione dei processori, predominano quattro approcci generali:

- **Condivisione del carico (Load sharing):** I processi non sono assegnati ad un particolare processore, si mantiene una coda globale di thread pronti, ed ogni processore, quando è in ozio, seleziona un thread dalla coda. Il termine *condivisione del carico* (*load sharing*) è usato per distinguere questa strategia dagli schemi a caricamento bilanciato (load balancing) nei quali il lavoro è allocato su una base più permanente (ad es., vedere [FEIT90a]).[1]
- **Gang scheduling:** Viene schedulato un insieme di thread correlati, da eseguire su un insieme di processori, sulla base di un processo per processore.
- **Assegnazione a processore dedicato:** Questo è l'opposto dell'approccio load-sharing, realizzando uno scheduling implicito definito dall'assegnazione dei thread ai processori. Ad ogni programma è allocato un numero di processori uguale al numero dei thread nel programma, per la durata dell'esecuzione del programma; quando il programma termina, i processori ritornano all'insieme generale per la possibile allocazione ad un altro programma.
- **Scheduling dinamico:** Il numero dei thread in un programma può essere cambiato durante il corso dell'esecuzione.

Condivisione del carico

La condivisione del carico è forse il più semplice approccio e quello che deriva più direttamente da un ambiente monoprocessore. Ha alcuni vantaggi:

- Il carico è distribuito uniformemente tra i processori, assicurandosi che nessun processore sia in ozio mentre c'è del lavoro da fare.
- Non è richiesto uno scheduler centralizzato: quando un processore è disponibile, la routine di scheduling del sistema operativo viene eseguita su quel processore, per selezionare il thread successivo.
- La coda globale può essere organizzata e vi si può accedere usando uno degli schemi discussi nel Capitolo 9, compresi gli schemi basati sulla priorità e gli schemi che considerano la storia dell'esecuzione o anticipano le richieste di elaborazione.

[LEUT90] analizza tre versioni diverse di condivisione del carico:

---

[1] La letteratura su quest'argomento si riferisce a questo approccio *come auto-scheduling (self-scheduling)*, dato che ogni processore schedula per se stesso senza considerare gli altri processori. Tuttavia, questo termine è anche usato nella letteratura per riferirsi a programmi scritti in un linguaggio che permette al programmatore di specificare lo scheduling (ad es. vedere [FOST91]).

- **First come first served (FCFS) (il primo processo che arriva è il primo che viene servito):** Quando arriva un job, i suoi thread sono posti consecutivamente alla fine della coda condivisa. Quando un processore termina il suo compito, prende il thread pronto successivo, e lo esegue fino al termine, o finché non si blocca.
- **Smallest number of thread first (vengono eseguiti prima i processi con il minor numero di thread):** La coda Ready condivisa è organizzata come una coda a priorità, con la più alta priorità data ai thread dei job con il più piccolo numero di thread non schedulati. I job di pari priorità sono ordinati a seconda di quale job arriva prima. Come per FCFS, un thread schedulato viene eseguito fino al termine, o finché non si blocca.
- **Preemptive smallest number of thread first (vengono eseguiti prima i processi con il minor numero di thread, con prerilascio):** La priorità più alta viene data ai job con il più piccolo numero di thread incompleti. Un job in arrivo con un numero di thread più piccolo di un job che è in esecuzione farà prerilasciare i thread che appartengono al job schedulato.

Usando modelli di simulazione, gli autori riportano che, su un'ampia gamma di caratteristiche del job, FCFS è superiore alle altre due politiche della lista precedente. Inoltre, gli autori trovano che una forma di gang scheduling, che vedremo nella prossima sottosezione, è generalmente superiore alla condivisione del carico.

Ci sono alcuni svantaggi nella condivisione del carico:

- La coda centrale occupa una regione di memoria cui si deve accedere in modo mutuamente esclusivo, perciò, potrebbe diventare un collo di bottiglia se molti processori cercano contemporaneamente del lavoro. Quando c'è solo un numero piccolo di processori, questo è un problema difficilmente rilevabile; tuttavia, quando il multiprocessore si compone di dozzine o anche centinaia di processori, la possibilità di un collo di bottiglia è reale.
- Thread prerilasciati difficilmente riprendono l'esecuzione sullo stesso processore; se ogni processore possiede una cache locale, l'operazione di caching diventa meno efficiente.
- Se i thread vanno trattati come un gruppo comune di thread, è improbabile che tutti i thread di un programma ottengano l'accesso ai processori allo stesso tempo. Se è richiesto un alto grado di coordinazione tra i thread di un programma, i cambi tra processi coinvolti possono seriamente compromettere le prestazioni.

Nonostante gli svantaggi potenziali, questo è uno degli schemi più comunemente usati nei multiprocessori correnti.

Un raffinamento della tecnica di condivisione del carico viene usato nel sistema operativo Mach [BLAC90, WEND89]. Il sistema operativo mantiene una coda di esecuzione locale per ogni processore, ed una coda di esecuzione globale condivisa: la coda in esecuzione locale è usata dai thread che sono stati temporaneamente legati ad un processore specifico. Un processore esamina prima la coda di esecuzione locale, per dare ai thread legati la preferenza assoluta sui thread non legati; come esempio di thread legati, uno o più processori potrebbero essere dedicati ad eseguire i processi che fanno parte del sistema operativo. Un altro esempio è che i thread di una particolare applicazione potrebbero essere distribuiti fra un certo numero di processori; con

l'aggiunta del software adeguato, questo fornisce supporto per il gang scheduling, discusso di seguito.

Gang scheduling

Il concetto di schedulare un insieme di processi simultaneamente su di un insieme di processori, è precedente all'uso dei thread. [JONE80] si riferisce al concetto come *scheduling di gruppo* (*group scheduling*) e cita i seguenti vantaggi:

- Se processi strettamente correlati sono eseguiti in parallelo, il blocco per sincronizzazione può essere ridotto, può essere necessario un minor cambio di processi e le prestazioni miglioreranno.
- L'overhead dello scheduling può essere ridotto perché una singola decisione riguarda un numero di processori e processi contemporaneamente.

Sul multiprocessore Cm*, viene usato il termine *coscheduling* [GEHR87]. Il coscheduling è basato sul concetto di schedulare un insieme di task collegati, chiamato task force. Gli elementi singoli di una task force tendono ad essere abbastanza piccoli, e sono quindi vicini all'idea di thread.

Il termine *gang scheduling* è stato applicato allo scheduling simultaneo dei thread che costituiscono un singolo processo [FEIT90b]; esso è necessario per le applicazioni con parallelismo da granularità media a granularità fine, le cui prestazioni scendono notevolmente quando una qualsiasi delle parti dell'applicazione non è in esecuzione, mentre le altre sono Ready, ed è anche utile per ogni applicazione parallela, anche per quelle che non sono troppo sensibili alle prestazioni. La necessità del gang scheduling è ampiamente riconosciuta, ed esistono implementazioni su una varietà di sistemi operativi multiprocessore.

Una maniera ovvia in cui il gang scheduling migliora le prestazioni della singola applicazione è che i cambi di processo sono ridotti al minimo. Supponiamo che un thread di un processo sia in esecuzione e raggiunga un punto in cui deve sincronizzarsi con un altro thread dello stesso processo; se l'altro thread non è in esecuzione, ma è già in coda Ready, il primo thread viene sospeso fino a quando non si può fare un cambio di processo su un qualsiasi altro processore, per caricare il thread necessario, e in un'applicazione con stretta coordinazione tra i thread tali cambi riducono drammaticamente le prestazioni. Lo scheduling simultaneo dei thread cooperanti può anche far risparmiare tempo nell'allocazione di risorse. Per esempio, thread gang-scheduled multipli possono accedere un file senza l'overhead aggiuntivo del lock durante un'operazione di ricerca o di lettura/scrittura.

L'uso di gang scheduling crea un requisito per l'allocazione dei processori. Una possibilità è la seguente. Supponiamo di avere N processori e M applicazioni, ciascuna delle quali ha N o meno thread. Allora potrebbe essere dato ad ogni applicazione 1/M del tempo disponibile su N processori, usando il time slicing. [FEIT90a] nota che questa strategia può essere inefficiente: si consideri un esempio in cui ci sono due applicazioni, una con quattro thread, l'altra con un thread. Usare l'allocazione uniforme del tempo (*uniform time allocation*) fa perdere il 37.5% delle risorse di processo, perché quando è in esecuzione l'applicazione con un singolo thread, tre processori sono lasciati in ozio (vedere Figura 10.2). Se ci sono diverse applicazioni con un

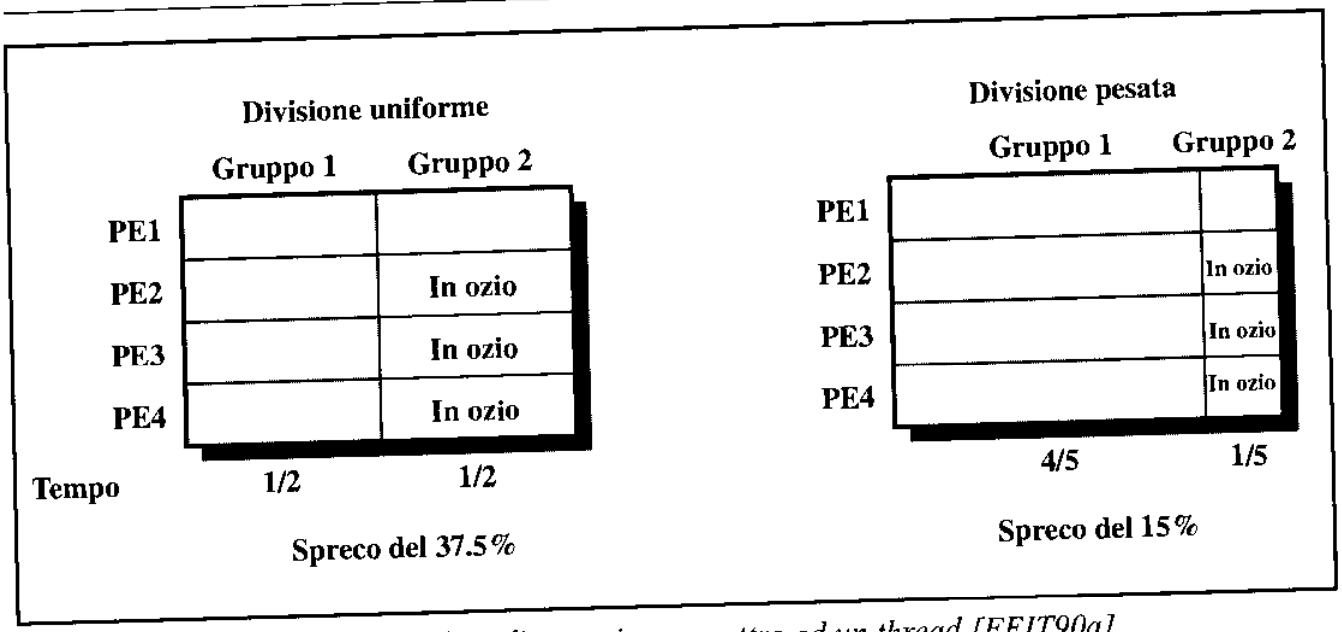


Figura 10.2 *Esempio di scheduling di gruppi con quattro od un thread [FEIT90a]*

thread, queste potrebbero essere allocate tutte insieme in modo da aumentare l'utilizzazione del processore; se questo non è possibile, un'alternativa allo scheduling uniforme è lo scheduling pesato dal numero di thread. Quindi, si potrebbe dare all'applicazione con quattro thread i quattro quinti del tempo, mentre all'applicazione con un thread solo si assegna un quinto del tempo, riducendo la perdita di tempo di processore al 15%.

Assegnazione di un processore dedicato

Una forma estrema di gang scheduling, suggerita in [TUCK89], è di dedicare un gruppo di processori ad un'applicazione per la durata dell'applicazione. Quando cioè un'applicazione viene schedulata, a ciascuno dei suoi thread viene assegnato un processore, che rimane dedicato a quel thread fino a che l'applicazione non termina l'esecuzione.

Può sembrare che quest'approccio sprechi il tempo del processore: se un thread di un'applicazione è bloccato in attesa di I/O, o per sincronizzarsi con un altro thread, allora il processore di quel thread rimane in ozio, non essendoci multiprogrammazione dei processori. In difesa di questa strategia si possono fare due osservazioni:

1. In un processore altamente parallelo, con decine o centinaia di processori, ciascuno dei quali rappresenta una piccola frazione del costo del sistema, l'utilizzo del processore non è più così importante come metro dell'efficienza o delle prestazioni.
2. Annullare totalmente i cambi di processo per tutto il tempo di esecuzione di un programma dovrebbe dare un sostanziale aumento di velocità a quel programma.

Sia [TUCK89] sia [ZAHO90] riportano analisi a supporto della seconda affermazione: la Figura 10.3 mostra i risultati di un esperimento ([TUCK89]). Gli autori hanno eseguito due applicazioni, una moltiplicazione tra matrici ed il calcolo di una Fast Fourier Transform (FFT), su un sistema a 16 processori. Ciascun'applicazione suddivide il suo problema in un numero di

task, che sono mappati sui thread che eseguono quell'applicazione. I programmi sono scritti in modo da usare un numero di thread variabile: in pratica, un'applicazione definisce ed accoda un certo numero di task. I task sono presi dalla coda e mappati sui thread disponibili dall'applicazione. Se ci sono meno thread rispetto ai task, allora i task lasciati in sospeso rimangono in coda e sono ricuperati dai thread non appena completano i task loro assegnati. Chiaramente, non tutte le applicazioni possono essere strutturate in questo modo, ma molti problemi numerici ed alcune altre applicazioni possono essere trattate in questo modo.

La Figura 10.3 mostra l'aumento di velocità delle applicazioni quando il numero dei thread che eseguono i task in ciascuna di esse varia tra 1 e 24. Consideriamo il caso in cui entrambe le applicazioni iniziano simultaneamente con 24 thread ciascuna; l'aumento di velocità, rispetto all'uso di un singolo thread per ogni applicazione, è 2.8 per la moltiplicazione tra matrici, e 2.4 per la FFT. La figura mostra che le prestazioni di entrambe le applicazioni peggiorano considerevolmente quando il numero dei thread in ogni applicazione supera 8, cioè quando il numero totale dei processi nel sistema supera il numero dei processori. Inoltre, maggiore è il numero dei thread, peggiori diventano le prestazioni, perché c'è una maggiore frequenza di prerilascio dei thread e nuovo scheduling. Quest'eccessivo prerilascio porta all'inefficienza per molte ragioni, tra cui il tempo trascorso aspettando che un thread sospeso lasci una sezione critica, il tempo perso nel cambio di processo ed il comportamento inefficiente della cache.

Gli autori concludono che una strategia efficace è quella di limitare il numero dei thread attivi al numero dei processori nel sistema. Se la maggior parte delle applicazioni sono a thread singolo, o possono usare la struttura di coda dei task, ciò fornirà un uso efficace e ragionevolmente efficiente delle risorse del processore.

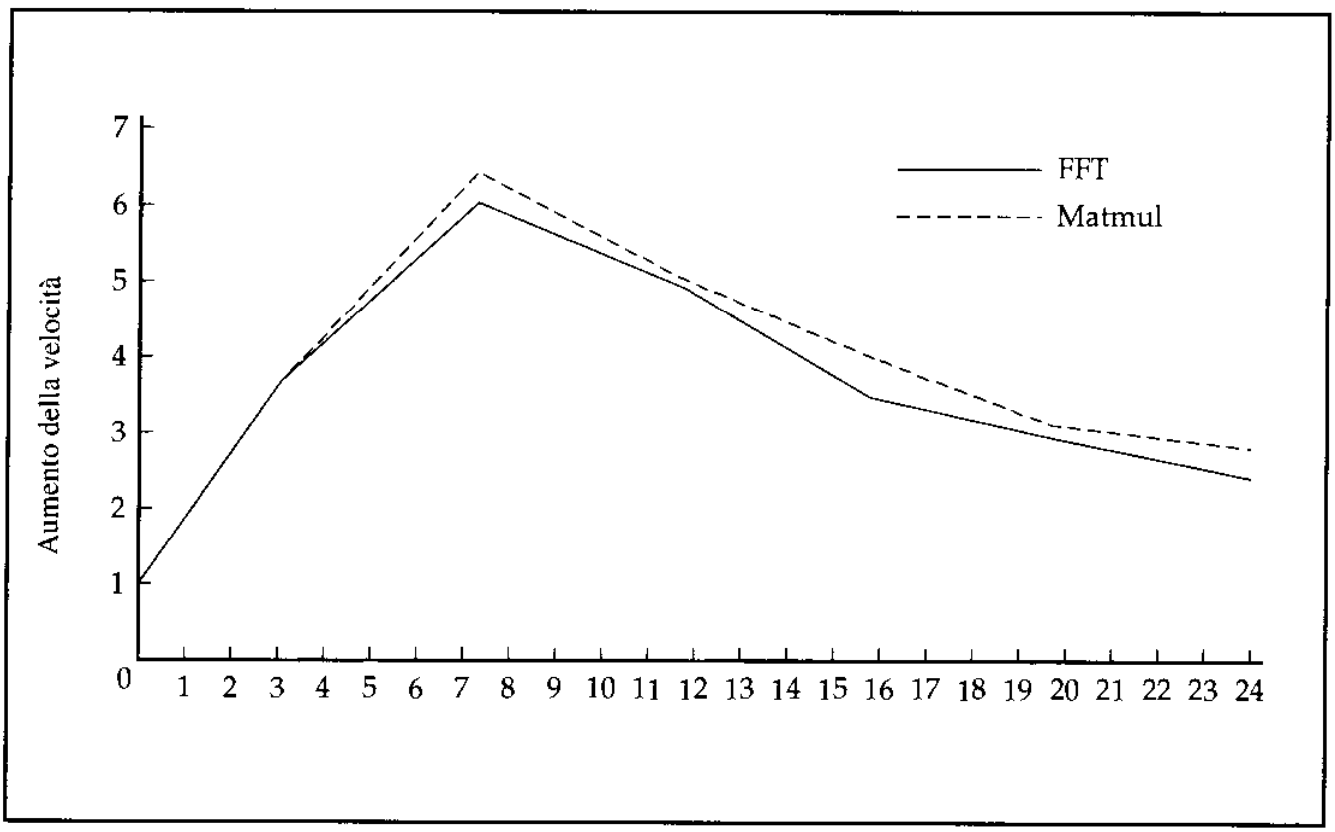


Figura 10.3 *Aumento di velocità dell'applicazione in funzione del numero dei processi [TUCK89]*

Sia l'assegnazione dedicata del processore, sia il gang scheduling affrontano il problema dello scheduling cominciando dalla questione dell'allocazione del processore. Si potrebbe osservare che il problema dell'allocazione del processore, su un multiprocessore, assomiglia più al problema dell'allocazione di memoria su un monoprocessore piuttosto che al problema dello scheduling su un monoprocessore. Nasce il problema di quanti processori assegnare ad un programma in ogni momento, che è simile al problema di quanti frame assegnare ad un dato processo in ogni momento. [GEHR87] propone il termine *activity working set (working set delle attività)*, analogo al working set della memoria virtuale, definito come il numero minimo di attività (thread) che devono essere schedulate simultaneamente sui processori perché l'applicazione progredisca in modo accettabile; come per gli schemi di gestione della memoria, non schedulare tutti gli elementi di un activity working set può portare al *thrashing del processore*. Questo accade quando lo scheduling dei thread i cui servizi sono richiesti, porta ad eliminare altri thread i cui servizi saranno presto necessari. Similmente, la frammentazione del processore (*processor fragmentation*) si riferisce ad una situazione in cui alcuni processori sono allocati, gli altri sono lasciati in sospeso, ma sono insufficienti di numero, oppure organizzati in modo non adatto a supportare i requisiti delle applicazioni in attesa. Per evitare questi problemi, il gang scheduling e l'allocazione di processori dedicati sono stati appositamente pensati.

Scheduling dinamico

Per alcune applicazioni, è possibile fornire linguaggi e strumenti di sistema che permettano al numero dei thread nel processo di essere modificati dinamicamente. Questo permetterebbe al sistema operativo di aggiustare il carico per migliorare l'utilizzo.

[ZAHO90] propone un approccio in cui il sistema operativo e l'applicazione collaborano nel prendere le decisioni di scheduling. Il sistema operativo è responsabile del partizionamento dei processori tra i job; ciascun job usa i processori correntemente nella sua partizione per eseguire alcuni sottoinsiemi dei suoi task eseguibili, mappando questi task nei thread. Una decisione adeguata di quale sottoinsieme eseguire, e di quale thread sospendere quando un processo è prerilasciato, viene lasciata alle applicazioni individuali (attraverso un insieme di routine di libreria a tempo di esecuzione). Quest'approccio può non essere adatto per tutte le applicazioni, tuttavia, alcune di esse potrebbero essere eseguite, per difetto, con un solo thread, mentre altre potrebbero essere programmate per trarre vantaggio da questa particolare caratteristica del sistema operativo.

In quest'approccio, la responsabilità di scheduling del sistema operativo viene limitata fondamentalmente all'allocazione dei processori e segue la seguente politica: quando un job richiede uno o più processori (o quando il job arriva per la prima volta, o perché cambiano i suoi requisiti),

1. Se ci sono processori in ozio, li usa per soddisfare la richiesta
2. Altrimenti, se il job che ha fatto la richiesta è un nuovo arrivato, gli alloca un solo processore, togliendolo da un job qualsiasi che n'abbia correntemente allocati più di uno.
3. Se una porzione qualsiasi della richiesta non può essere soddisfatta, rimane in sospeso fino a quando un processore diventa disponibile per essa, o il job annulla la richiesta (ad es., se non c'è più bisogno di processori extra).

Al rilascio di uno o più processori (compresa la terminazione del job),

4. Scandisce la coda corrente delle richieste insoddisfatte dei processori; assegna un solo processore ad ogni job nella lista che non ha correntemente processori (cioè, a tutti i nuovi arrivati in attesa), poi scandisce nuovamente la lista, allocando il resto dei processori su base FCFS.

Le analisi riportate in [ZAHO90] e [MAJU88] suggeriscono che per le applicazioni che possono trarre vantaggio dallo scheduling dinamico, quest'approccio è superiore rispetto al gang scheduling o all'assegnazione di processori dedicati. Tuttavia l'overhead di quest'approccio può annullare questo apparente vantaggio nelle prestazioni. È necessario fare esperienza con sistemi reali per provare il valore dello scheduling dinamico.

10.2 Scheduling in tempo reale

Premessa

L'elaborazione in tempo reale (real time) sta diventando una disciplina sempre più importante, ed il sistema operativo, ed in particolare lo scheduler, è forse il componente più importante di un sistema in tempo reale. Esempi di applicazioni attuali dei sistemi in tempo reale sono: il controllo degli esperimenti di laboratorio, impianti di controllo di processo, robotica, controllo del traffico aereo, telecomunicazioni, e sistemi di controllo e comando militari. I sistemi della prossima generazione includeranno la guida automatica, i controllori di robot con giunture elastiche, sistemi di produzione intelligenti, stazioni spaziali ed esplorazioni sottomarine.

L'elaborazione in tempo reale può essere definita come quel tipo di elaborazione in cui la correttezza del sistema dipende non solo dal risultato logico dell'elaborazione, ma anche dal tempo a cui i risultati sono stati prodotti. Possiamo definire un sistema in tempo reale definendo cosa si intende con un processo od un task in tempo reale[2]. In generale, in un sistema in tempo reale, solo alcuni dei task sono task in tempo reale, e questi hanno un certo grado di urgenza: tali task tentano di controllare o di reagire ad eventi, che avvengono nel mondo esterno. Poiché questi eventi accadono "in tempo reale", un task in tempo reale deve essere in grado di mantenere i rapporti con gli eventi che lo riguardano. Perciò, è possibile di solito associare una scadenza (*deadline*) ad un particolare task, dove la scadenza specifica un tempo di inizio od un tempo di completamento. Un tale task può essere classificato come rigido (hard) o morbido (soft). Un task **hard real time** è quello che deve rispettare la sua scadenza, altrimenti causerà danni indesiderabili, o errori fatali al sistema; un task **soft real time** ha una scadenza associata che è

[2] Come al solito, la terminologia crea un problema, dato che varie parole sono usate in letteratura con vari significati. Comunemente, un processo particolare opera sotto vincoli in tempo reale di natura ripetitiva, cioè, il processo dura per molto tempo e, durante quel tempo, esegue alcune funzioni ripetitive in risposta ad eventi in tempo reale. Riferiamoci, per questa sezione, ad una funzione individuale, chiamandola task. Perciò, il processo può essere visto progredire per mezzo di una sequenza di thread; in ogni momento, il processo è impegnato in un solo task, ed è il processo/task che deve essere schedulato.

desiderabile, ma non è obbligatoria; è ancora sensato schedulare e completare il task anche se ha passato la sua scadenza.

Un'altra caratteristica dei task in tempo reale è che possono essere periodici od aperiodici. Un **task aperiodico** ha una scadenza, entro la quale deve iniziare o terminare, ma potrebbe avere un vincolo sia sul tempo di partenza sia su quello di fine; nel caso di un **task periodico**, il requisito può essere stabilito come "una volta ogni T" o "esattamente dopo T unità".

Caratteristiche dei sistemi operativi in tempo reale

I sistemi operativi in tempo reale possono essere caratterizzati con requisiti unici in cinque aree generali [MORG92]:

- Determinismo
- Prontezza (responsiveness)
- Controllo utente
- Affidabilità
- Operatività fail-soft.

Un sistema operativo è **deterministico** nel senso che effettua operazioni a tempi fissati e predeterminati, oppure entro intervalli di tempo predeterminati. Quando molti processi competono per le risorse e per il tempo di processore, nessun sistema può essere completamente deterministico; in un sistema operativo in tempo reale, le richieste di servizio del processo sono dettate da tempi ed eventi esterni. Il limite entro cui un sistema operativo può soddisfare deterministicamente le richieste dipende per prima cosa dalla velocità con cui può rispondere alle interruzioni e, secondariamente, dalla capacità o incapacità del sistema di gestire tutte le richieste entro il tempo richiesto.

Una misura utile dell'abilità di un sistema operativo nel funzionare deterministicamente, è il ritardo massimo, da quando arriva un'interruzione da un dispositivo ad alta priorità, a quando inizia il servizio dello stesso. In sistemi operativi non in tempo reale, questo ritardo può essere di decine o centinaia di millisecondi, mentre in un sistema operativo in tempo reale questo ritardo può avere un limite massimo di pochi microsecondi, fino a un millisecondo.

Una caratteristica collegata, ma distinta, è la **prontezza**. Il determinismo riguarda il ritardo del sistema operativo prima di riconoscere un'interruzione; la prontezza riguarda il tempo impiegato, dopo il riconoscimento, per servire l'interruzione. Le caratteristiche della prontezza sono le seguenti:

1. La quantità di tempo richiesta per gestire inizialmente l'interruzione ed iniziare l'esecuzione della routine di servizio dell'interruzione (ISR = Interrupt Service Routine). Se l'esecuzione dell'ISR richiede un cambio di processo, allora il ritardo sarà più lungo rispetto ad eseguire l'ISR entro il contesto del processo corrente.
2. La quantità di tempo richiesto per eseguire l'ISR, generalmente dipendente dalla piattaforma hardware.

3. L'effetto dell'annidamento di interruzioni. Se un'ISR può essere interrotta dall'arrivo di un altra interruzione, allora il servizio sarà ritardato.

Il determinismo e la prontezza costituiscono insieme il tempo di risposta agli eventi esterni. I requisiti del tempo di risposta sono critici per i sistemi in tempo reale, perché tali sistemi devono scontrarsi con i requisiti temporali imposti dagli utenti, dai dispositivi e da flussi di dati esterni al sistema.

Il **controllo ntente** generalmente è più ampio in un sistema operativo in tempo reale che nei sistemi operativi ordinari. In un tipico sistema operativo non in tempo reale, l'utente o non ha il controllo sulle funzioni di scheduling del sistema operativo, o può solo fornire un'informazione di alto livello, come ad esempio raggruppare utenti in più di una classe di priorità. Nei sistemi in tempo reale, tuttavia, è essenziale permettere all'utente un controllo a granularità fine sulla priorità dei task: l'utente dovrebbe essere in grado di distinguere tra task hard e soft real time, e di specificare le priorità relative all'interno di ciascuna classe. Un sistema in tempo reale potrebbe permettere anche all'utente di specificare caratteristiche come l'uso della paginazione, o il trasferimento su disco dei processi, quali processi devono sempre essere presenti nella memoria principale, quali algoritmi di trasferimento disco devono essere usati, quali diritti hanno i processi nelle varie bande di priorità, e così via.

L'**affidabilità** è tipicamente molto più importante per i sistemi in tempo reale che per i sistemi non in tempo reale. Un fallimento transitorio in un sistema non in tempo reale può essere risolto semplicemente effettuando un reboot del sistema; un fallimento di un processore in un sistema multiprocessore non in tempo reale può comportare un livello di servizio ridotto fino alla riparazione od alla sostituzione del processore fallito. Ma un sistema in tempo reale controlla e risponde ad eventi in tempo reale, e la perdita o il degrado delle prestazioni possono avere conseguenze catastrofiche, che vanno da una perdita finanziaria, a un grave danno agli strumenti, fino alla perdita della vita.

Come in altre aree, la differenza tra un sistema in tempo reale ed un sistema non in tempo reale è una differenza di grado: anche un sistema in tempo reale deve essere progettato per rispondere a diversi modi di fallimento. L'**operatività fail-soft** si riferisce all'abilità di un sistema di fallire in modo da preservare la quantità maggiore possibile di capacità e dati. Per esempio, un tipico sistema UNIX tradizionale, quando trova dati corrotti all'interno del kernel, lancia un messaggio di errore sulla console del sistema, memorizza il contenuto della memoria principale su disco, per successive analisi dell'errore, e termina l'esecuzione del sistema. Viceversa, un sistema in tempo reale cercherà o di correggere il problema, o di minimizzare i suoi effetti, mentre l'esecuzione continua; solitamente, il sistema notifica ad un utente o ad un processo utente che dovrebbe cercare di compiere un'azione correttiva, poi continua l'operazione, eventualmente ad un livello di servizio ridotto. Nel caso in cui sia necessario uno spegnimento, si fa un tentativo per mantenere la consistenza dei dati e del file.

Un importante aspetto dell'operatività fail-soft è la *stabilità*: un sistema in tempo reale è stabile se, nei casi in cui è impossibile soddisfare tutte le scadenze dei task, il sistema soddisferà le scadenze dei task più critici e a più alta priorità, anche se alcune scadenze di task meno critici non vengono sempre soddisfatte.

Per soddisfare i precedenti requisiti, i sistemi operativi in tempo reale di oggi, normalmente comprendono le seguenti caratteristiche [STAN89]:

- Veloce cambio di thread o di processo
- Piccola dimensione (con associata funzionalità minima)
- Capacità di rispondere velocemente ad interruzioni esterne
- Multitasking con strumenti di comunicazione tra processi come ad esempio semafori, segnali ed eventi
- Uso di speciali file sequenziali che possono accumulare dati ad alta velocità
- Scheduling con prerilascio basato sulla priorità
- Minimizzazione degli intervalli in cui le interruzioni sono disabilitate
- Primitive per ritardare i task di un tempo fissato e per mettere in pausa o rimettere in esecuzione i task
- Allarmi speciali e timeout.

Il cuore di un sistema in tempo reale è lo scheduler a breve termine dei task. Nella progettazione di un tale scheduler, l'equità (fairness) e la minimizzazione del tempo di risposta medio non sono importanti; ciò che è importante è che tutti i task hard real time terminino (od inizino) alla loro scadenza, e che anche quanti più possibili task soft real time terminino (od inizino) alla loro scadenza.

La maggior parte dei sistemi in tempo reale contemporanei, non sono in grado di gestire direttamente le scadenze; sono invece progettati per essere il più pronti possibile ai task in tempo reale, in modo che, quando si approssima una scadenza, un task possa essere velocemente schedulato. Da questo punto di vista, le applicazioni in tempo reale tipicamente richiedono tempi di risposta deterministici in un intervallo da qualche millisecondo a meno di un millisecondo, per un vasto insieme di condizioni; applicazioni avanzate - in simulatori per velivoli militari, per esempio - spesso hanno vincoli fra 10 e 100 μs [ATLA89].

La Figura 10.4 illustra una gamma di possibilità. In uno scheduler con prerilascio che usa un semplice scheduling round-robin, un task in tempo reale sarebbe aggiunto alla coda Ready per attendere il suo quanto di tempo successivo, come illustrato nella Figura 10.4a: in questo caso, il tempo di scheduling sarà generalmente inaccettabile per applicazioni in tempo reale. In alternativa, in uno scheduler senza prerilascio, potremmo usare un meccanismo di scheduling a priorità, dando ai task in tempo reale la priorità maggiore: in questo caso, un task in tempo reale che è Ready, sarebbe schedulato non appena il processo corrente si blocca o termina l'esecuzione (Figura 10.4b). Questo potrebbe portare ad un ritardo di alcuni secondi, se un task lento a bassa priorità fosse in esecuzione in un momento critico: anche questo approccio non è accettabile. Un approccio più promettente è combinare le priorità con le interruzioni di clock, in modo che punti di prerilascio avvengano ad intervalli regolari: quando avviene un punto di prerilascio, il task correntemente in esecuzione viene prerilasciato se un task a maggiore priorità sta aspettando: questo potrebbe capitare anche ai task che sono parte del kernel del sistema operativo. Un tale ritardo può essere dell'ordine di alcuni millisecondi (Figura 10.4c): può essere adeguato per alcune applicazioni in tempo reale, ma non è ancora adeguato per applicazioni più esigenti. In tali casi, si usa un approccio chiamato prerilascio immediato: il sistema operativo risponde alle

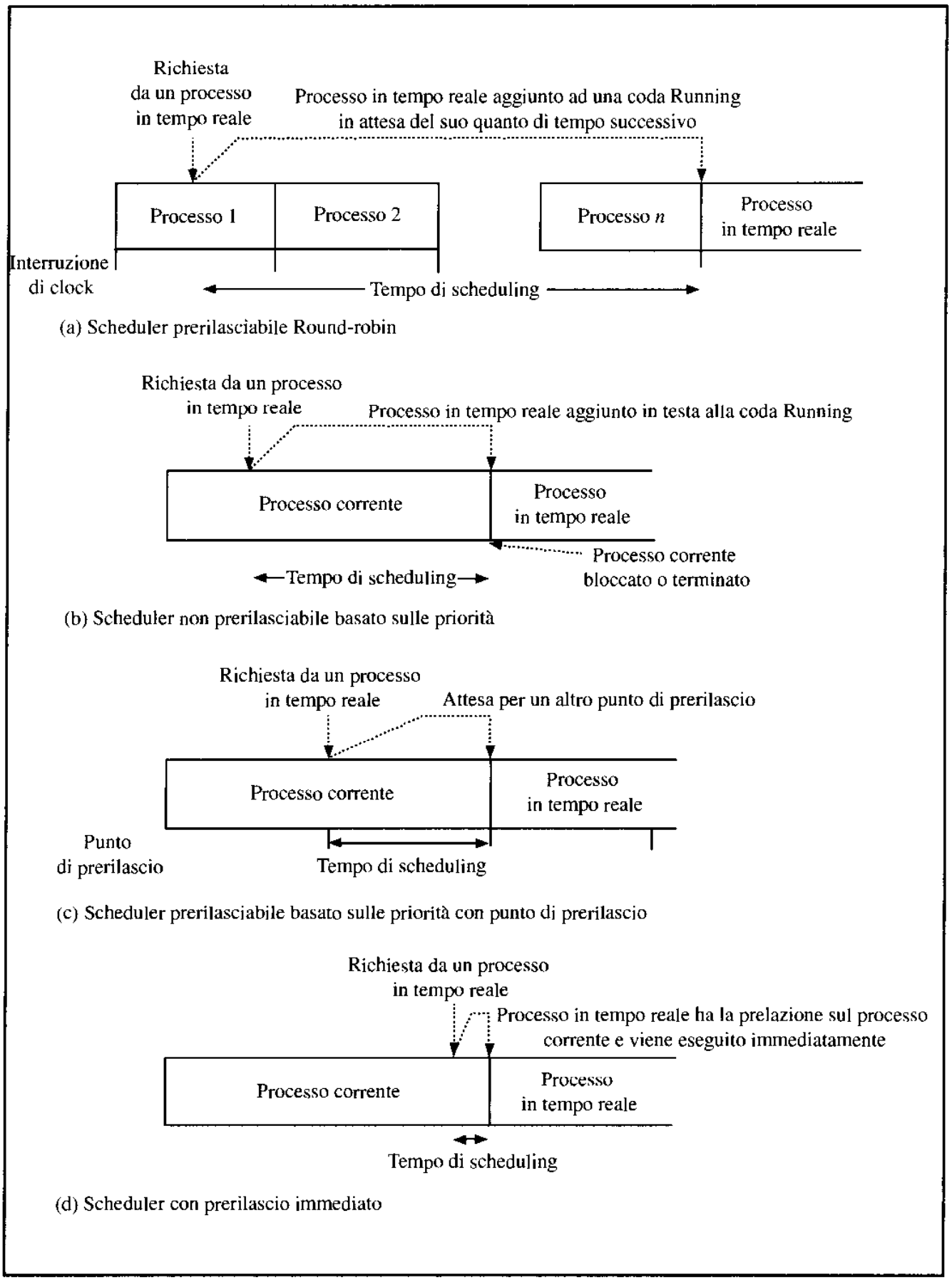


Figura 10.4 *Scheduling di un processo in tempo reale*

interruzioni quasi immediatamente, a meno che il sistema non sia in una sezione critica. I ritardi di scheduling per un task in tempo reale possono quindi ridursi fino a 100 μs o meno.

Scheduling in tempo reale

Lo scheduling in tempo reale è una delle aree di ricerca più attive dell'informatica. In questa sottosezione, si presenta una panoramica dei vari approcci allo scheduling in tempo reale e si esaminano due classi celebri di algoritmi di scheduling.

In una rassegna di algoritmi di scheduling in tempo reale, [RAMA94] osserva che i vari approcci di scheduling dipendono da (1) se un sistema effettua analisi di schedulabilità, (2) se le fa staticamente o dinamicamente, e (3) se il risultato dell'analisi stessa produce uno scadenzario o un piano seguendo il quale i task vengono allocati a tempo d'esecuzione. Basandosi su queste considerazioni, gli autori identificano le seguenti classi di algoritmi:

- **Approcci statici basati su tabelle:** Questi algoritmi effettuano un'analisi statica delle possibili scadenze di allocazione; il risultato di un'analisi è uno scadenzario che determina, a tempo d'esecuzione, quando un task deve iniziare l'esecuzione.
- **Approcci statici con prerilascio basati su priorità:** Si effettua nuovamente un'analisi statica, ma nessuno scadenzario viene redatto: l'analisi serve per assegnare le priorità ai task, in modo da poter usare un tradizionale scheduler con prerilascio basato su priorità.
- **Approcci dinamici basati sulla pianificazione:** La fattibilità è determinata a tempo d'esecuzione (dinamicamente) piuttosto che prima dell'inizio dell'esecuzione (staticamente): un task in arrivo è accettato per l'esecuzione solo se è possibile far fronte ai suoi vincoli temporali. Uno dei risultati dell'analisi di fattibilità è uno scadenzario, od un piano, usato per decidere quando allocare questo task.
- **Approcci dinamici del migliore sforzo:** Non è effettuata nessuna analisi di fattibilità: il sistema cerca di soddisfare tutti le scadenze, ed abortisce ogni processo iniziato che manca la propria scadenza.

Lo **scheduling statico basato su tabelle** è applicabile ai task periodici; l'input dell'analisi è il tempo d'arrivo periodico, il tempo d'esecuzione, le scadenze di terminazione periodiche e la priorità relativa di ogni task. Lo scheduler tenta di sviluppare uno scadenzario con cui si possa far fronte ai requisiti di tutti i task periodici. Questo è un approccio prevedibile, ma inflessibile, perché ogni cambiamento ai requisiti dei task richiede di rifare lo scadenzario. La tecnica earliest-deadline-first (che privilegia i task che hanno la scadenza prima in ordine temporale) od altre tecniche di scadenze periodiche (che discuteremo più avanti) sono tipiche di questa categoria di algoritmi di scheduling.

Lo **scheduling statico con prerilascio basato su priorità** utilizza il meccanismo di scheduling con prerilascio basato su priorità, tipico di molti sistemi in multiprogrammazione ma non in tempo reale. In un sistema non in tempo reale, per determinare la priorità si possono usare una varietà di fattori: ad esempio, in un sistema time-sharing, la priorità di un processo cambia a secondo se esso è processor bound oppure I/O bound. In un sistema in tempo reale, l'assegnazione della priorità è legata ai vincoli temporali associati ad ogni task: un esempio di questo

approccio è l'algoritmo a frequenza monotona (discusso più avanti), che assegna priorità statiche ai task basate sui loro periodi.

Con **lo scheduling dinamico basato sulla pianificazione** dopo che è arrivato un task, ma prima che inizi la sua esecuzione, si fa un tentativo di creare uno scadenzario, che contenga i task precedentemente schedulati oltre al nuovo arrivato. Se quest'ultimo può essere schedulato in modo tale che le sue scadenze siano soddisfatte, e che nessun task già schedulato manchi una scadenza, allora lo scadenzario viene aggiornato con il nuovo arrivato

Lo **scheduling dinamico del migliore sforzo** è l'approccio usato da molti sistemi in tempo reale che sono attualmente disponibili in commercio: quando arriva un task, il sistema gli assegna una priorità basata sulle caratteristiche del task. Solitamente si usa uno scheduling con scadenza, come lo scheduling earliest-deadline. Tipicamente i task sono aperiodici, così non è possibile nessuna analisi statica di scheduling: con questo tipo di scheduling, fino a quando non arriva una scadenza oppure il task termina, non sappiamo se un vincolo temporale verrà rispettato. Questo è lo svantaggio maggiore di questa forma di scheduling; il suo vantaggio è che è facile da implementare.

Scheduling con scadenza

La maggior parte dei sistemi operativi contemporanei è progettata per iniziare i task in tempo reale il più rapidamente possibile, e quindi enfatizza la velocità nella gestione delle interruzioni e nell'allocazione dei task. Di fatto, questa non è un metro particolarmente utile nella valutazione dei sistemi operativi in tempo reale: generalmente alle applicazioni in tempo reale non interessa la velocità pura, ma piuttosto il completamento (o l'inizio) dei task al tempo "giusto": né troppo presto, né troppo tardi, nonostante vi possano essere richieste di risorse dinamiche e conflitti, sovraccarico di elaborazione e fallimenti hardware o software. Ne segue che le priorità forniscono uno strumento grezzo e non catturano il requisito del completamento (od inizio) al tempo giusto.

Recentemente, ci sono state varie proposte per approcci allo scheduling dei task in tempo reale più potenti ed adeguati, tutti basati sull'avere informazioni supplementari su ciascun task. Nella sua forma più generale, si potrebbero usare le seguenti informazioni su ciascun task:

- **Tempo di Ready:** Tempo in cui il task diventa Ready; nel caso di un task ripetitivo o periodico, questo è effettivamente una successione di tempi che è nota in anticipo. Nel caso di un task aperiodico, questo tempo può essere noto in anticipo, oppure il sistema operativo può solo esserne a conoscenza quando il task è effettivamente pronto.
- **Scadenza di inizio:** Tempo entro cui il task deve essere iniziato.
- **Scadenza di completamento**: Tempo a cui il task deve essere completato. Una tipica applicazione in tempo reale potrà avere o scadenza di inizio o scadenza di completamento, ma non entrambe.
- **Tempo di elaborazione:** Tempo richiesto per eseguire il task fino al termine. In alcuni casi questo è fornito; in altri, il sistema operativo misura una media esponenziale; per altri sistemi di scheduling, questa informazione non viene usata.

- **Requisiti di risorse:** Insieme di risorse (diverse dal processore) richieste dal task mentre sta eseguendo.
- **Priorità:** Misura l'importanza relativa del task. I task hard real time possono avere una priorità "assoluta": il sistema fallisce se viene mancata una scadenza. Se il sistema deve continuare ad eseguire, qualsiasi cosa succeda, allora sia ai task hard real time, sia ai task soft real time, si possono assegnare priorità relative, come indicazioni allo scheduler.
- **Struttura di subtask:** Un task può essere decomposto in un subtask obbligatorio ed un subtask opzionale. Solo il subtask obbligatorio possiede una scadenza rigida.

Ci sono alcune decisioni da prendere, nello scheduling in tempo reale, quando sono prese in considerazione le scadenze: quale task schedulare successivamente e quale genere di prerilascio è consentito. Si può dimostrare, per una data strategia di prerilascio ed usando scadenza di inizio o di completamento, che una politica di schedulazione del task earliest-deadline minimizza la percentuale di task che mancano la loro scadenza [HONG89, PANW88]. Questa conclusione vale sia per configurazioni a processore singolo sia per multiprocessori.

L'altro problema critico di progetto è quello del prerilascio. Quando sono specificate scadenze di inizio, allora uno scheduler senza prerilascio non ha senso; in questo caso, il task in tempo reale avrebbe la responsabilità di bloccarsi, dopo il completamento della parte obbligatoria o critica della sua esecuzione, permettendo alle altre scadenze di inizio di essere soddisfatte. Questo è adatto allo schema della Figura 10.4b. Per un sistema con scadenza di completamento, una strategia con prerilascio (Figura 10.4c o 10.4d) è più adeguata: per esempio, se il task X è in esecuzione ed il task Y è pronto, ci possono essere circostanze in cui il solo modo per permettere a tutti e due di far fronte alle loro scadenze di completamento è di prerilasciare X, eseguire Y fino al termine e poi riprendere X fino al termine.

Come esempio di scheduling di task periodici con scadenza di completamento, consideriamo un sistema che raccoglie ed elabora dati da due sensori, A e B. La scadenza per la raccolta dei dati dal sensore A arriva ogni 20 ms, e quella per B ogni 50 ms; contando l'overhead del sistema operativo, elaborare ogni campione di dati da A richiede 10 ms, e 25 ms elaborare ogni campione di dati da B. La Tabella 10.2 riassume il profilo dell'esecuzione dei due task.

Il computer è in grado di prendere una decisione di scheduling ogni 10 ms: in tale ipotesi, supponiamo di voler usare uno schema di scheduling con priorità: i primi due diagrammi temporali di Figura 10.5 mostrano il risultato. Se A ha la priorità più alta, alla prima attivazione di B vengono assegnati solo 20 ms di tempo di elaborazione, in due pezzi da 10 ms, prima del tempo di scadenza, e quindi B fallisce. Se viene data a B la priorità più alta, allora A manca la sua prima scadenza; il diagramma temporale finale mostra l'uso dello scheduling earliest-deadline. Al tempo $t = 0$, arrivano sia A1 sia B1; poiché A1 ha la scadenza più vicina, viene schedulato prima; e quando A1 ha terminato, il processore passa a B1. A $t = 20$, arriva A2: poiché A2 ha una scadenza più vicina rispetto a B1, B1 viene interrotto affinché A2 possa essere eseguito fino al termine. Al termine di A2, B1 viene ripreso, al tempo $t = 30$. A $t = 40$, arriva A3, ma questa volta la scadenza di completamento di B1 è la più vicina, e può essere eseguito fino al suo completamento, a $t = 45$. Il processore passa ad A3, che finisce al tempo $t = 55$.

In questo esempio, lo scheduler che da priorità, a qualsiasi punto di prerilascio, al task con la scadenza più vicina può soddisfare i requisiti del sistema; poiché i task sono periodici e prevedibili,

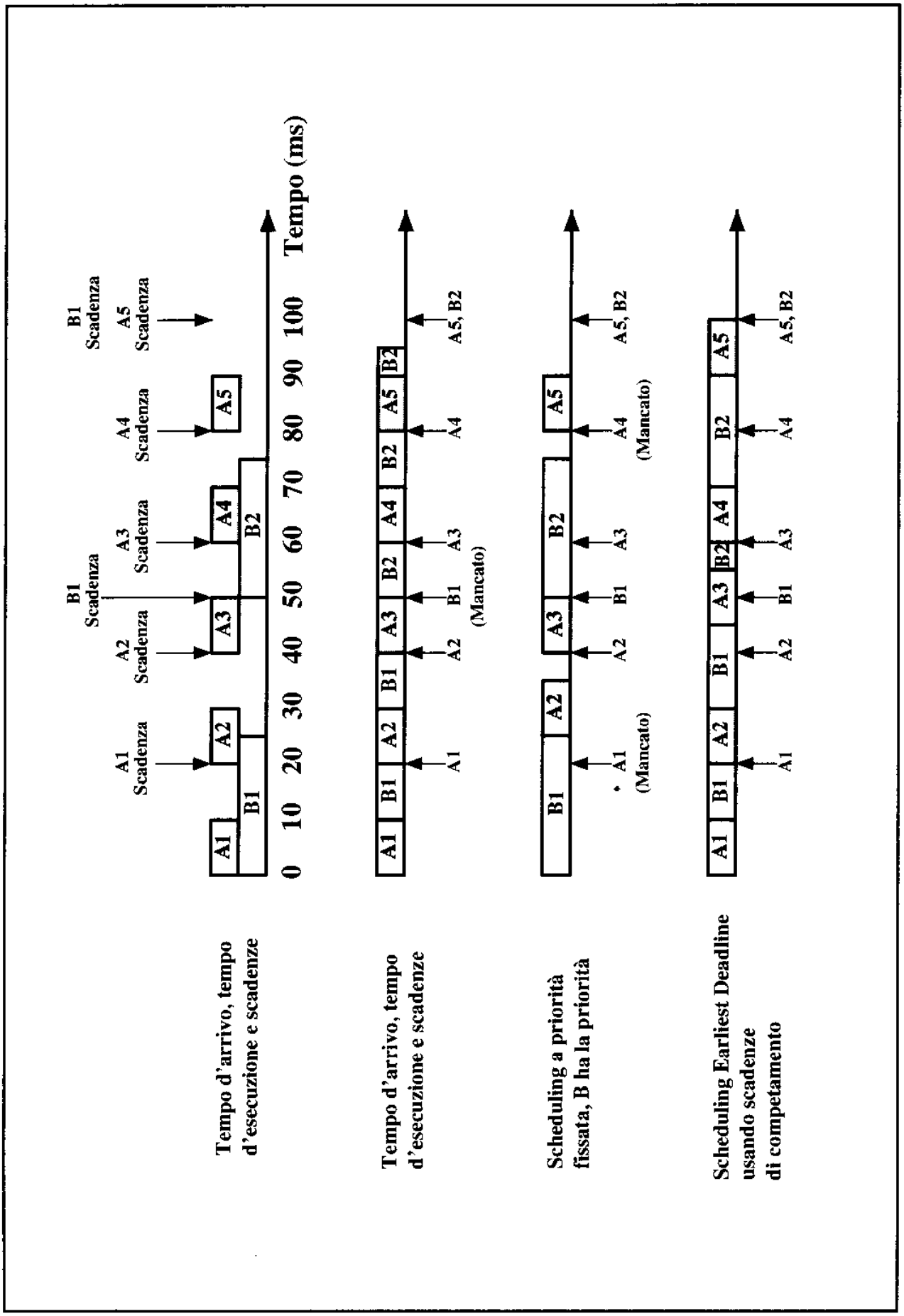


Figura 10.5 *Scheduling di task in tempo reale periodici con scadenza di completamento*

Tabella 10.2 *Profilo dell'esecuzione di due task periodici*

Processo	Tempo d'arrivo	Tempo d'esecuzione	Scadenza di inizio
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	60
A(5)	80	10	100
...	...	...	...
B(1)	0	25	50
B(2)	50	25	100
...	...	...	...

viene usato un approccio di scheduling statico basato su tabelle.

Consideriamo ora uno schema per trattare i task aperiodici con scadenza di inizio: la parte superiore della Figura 10.6 mostra i tempi di arrivo e le scadenze di inizio per un esempio costituito da cinque task, ciascuno dei quali possiede un tempo di esecuzione di 20 ms. La Tabella 10.3 riassume il profilo dell'esecuzione dei cinque task.

Uno schema semplice consiste nello schedulare sempre il task Ready con earliest-deadline e lasciare che quel task sia eseguito fino al termine; ma quando si usa questo approccio nell'esempio di Figura 10.6, si nota che, sebbene il task B richieda un servizio immediato, il servizio stesso gli è negato. Questo è il rischio nel trattare i task aperiodici, specialmente con scadenza di inizio: un raffinamento della politica migliorerebbe le prestazioni se le scadenze fossero conosciute prima che il task sia Ready. Questo sistema, chiamato earliest-deadline con tempi in ozio non forzati, opera come segue: schedula sempre il task ammissibile con scadenza più vicina e lo

Tabella 10.3 *Profilo dell'esecuzione di cinque task aperiodici*

Processo	Tempo d'arrivo	Tempo d'esecuzione	Scadenza di inizio
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

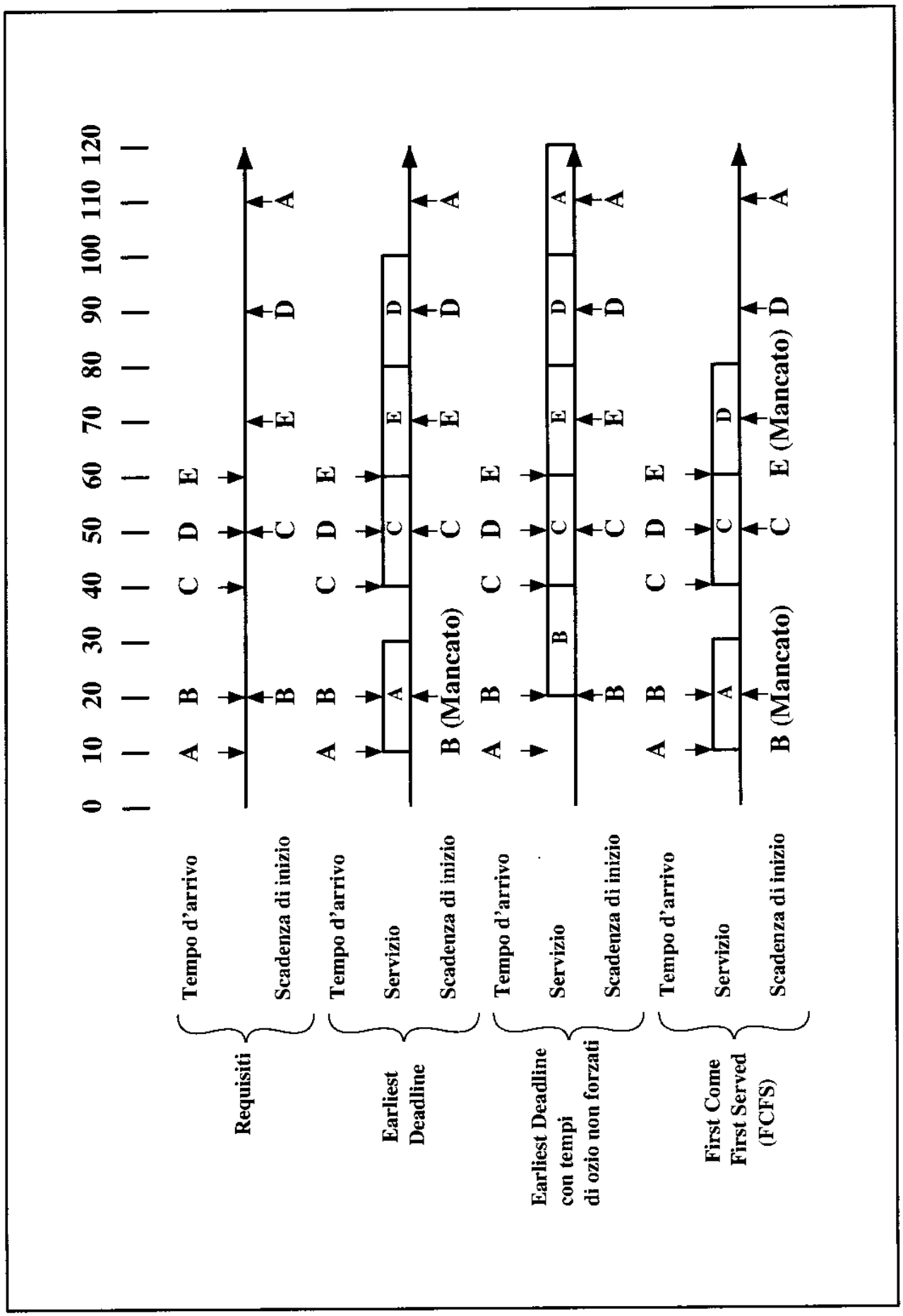


Figura 10.6 *Scheduling di task in tempo reale aperiodici con scadenza di inizio*

lascia in esecuzione fino al termine. Se nessun task ammissibile fosse Ready, con questo sistema il processore rimane in ozio, anche se ci sono altri task Ready: si nota, nel nostro esempio, che il sistema si astiene dallo schedulare il task A, anche se è l'unico task Ready. Il risultato è che, anche se il processore non è usato alla massima efficienza, tutti i requisiti di scheduling sono soddisfatti. Infine mostriamo, per confronto, il sistema FCFS: in questo caso, i task B ed E non soddisfano le loro scadenze.

Scheduling a frequenza monotona

Uno dei metodi più promettenti per risolvere i conflitti di scheduling multitask per task periodici è lo scheduling a frequenza monotona (RMS, Rate Monotonic Scheduling). Lo schema fu proposto tempo fa in [LIU73], ma ha guadagnato solo recentemente popolarità ([SHA94]): RMS assegna le priorità ai task sulla base dei loro periodi.

La Figura 10.7 illustra i parametri rilevanti per i task periodici: il periodo del task, T, è la quantità di tempo compresa tra l'arrivo di un'istanza del task e l'arrivo dell'istanza successiva del task. La frequenza di un task (in Hertz), è semplicemente l'inverso del suo periodo (in secondi). Per esempio, un task con un periodo di 50 ms arriva alla frequenza di 20 Hz; tipicamente, la fine del periodo di un task è anche la scadenza rigida del task, sebbene alcuni task possano avere scadenze più prossime. Il tempo d'esecuzione (o di calcolo) C, è la quantità di tempo di elaborazione richiesta da ogni occorrenza del task. Dovrebbe essere chiaro che in un sistema monoprocessore, il tempo d'esecuzione non deve essere più grande del periodo (si deve avere $C \leq T$). Se un task periodico viene eseguito sempre fino al termine - cioè, se a nessuna istanza del task non viene mai negato il servizio a causa di insufficienza di risorse - allora l'utilizzo del processore da parte di questo task è $U = C/T$. Per esempio, se un task ha un periodo di 80 ms ed un tempo d'esecuzione di 55 ms, il suo utilizzo del processore è 55 / 80 = 0.6875.

Per RMS, il task con la priorità più alta è quello con il periodo più breve, il secondo task con la priorità più alta è quello con il secondo periodo più breve, e così via; quando più di un task è Ready, viene servito prima quello con la priorità più alta. Se si fa un grafico della priorità dei task in funzione della loro frequenza, il risultato è una funzione monotona crescente (Figura 10.8); da qui il nome, scheduling a frequenza monotona.

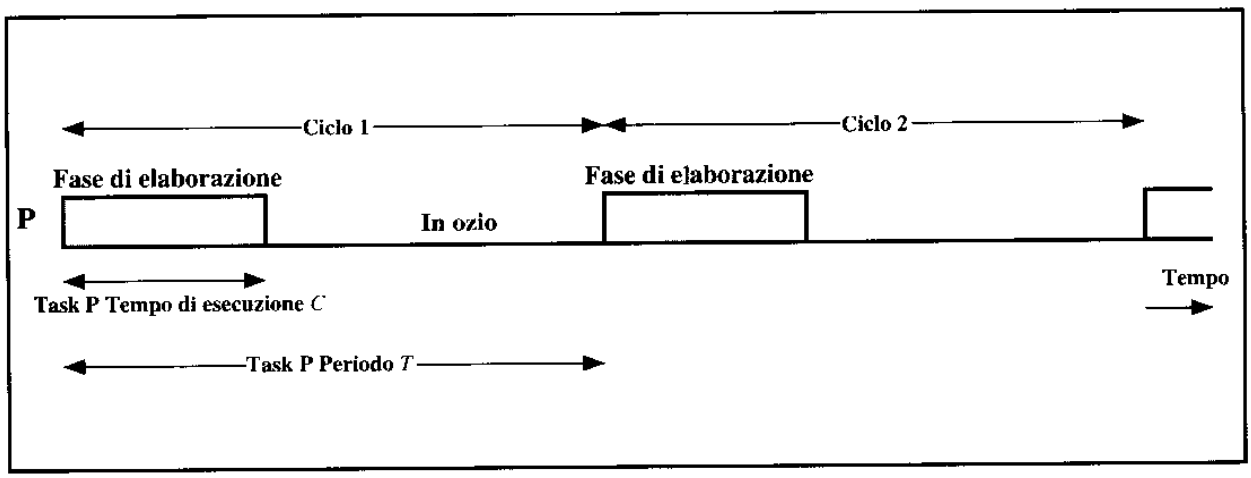


Figura 10.7 *Diagramma temporale di un task periodico*

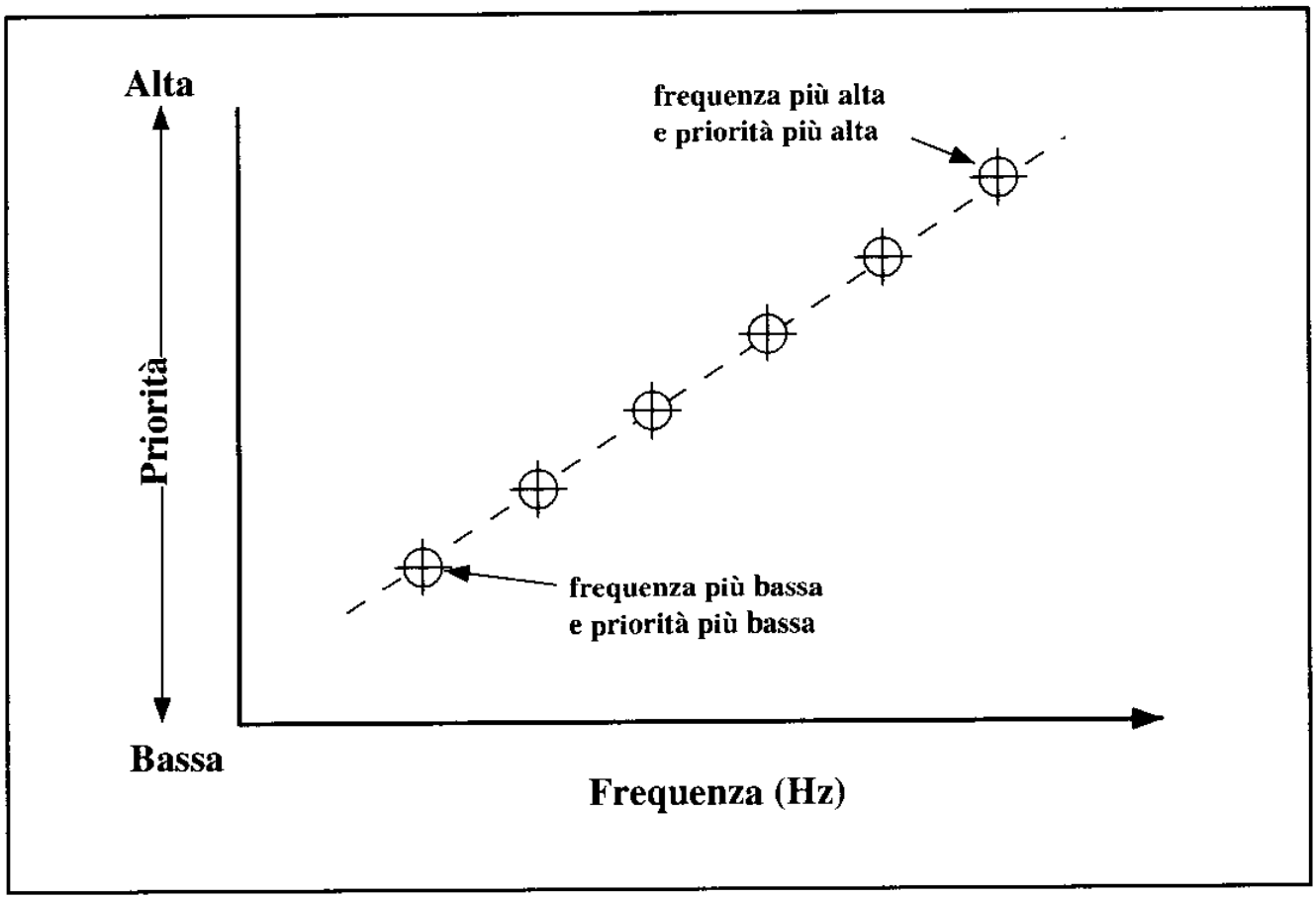


Figura 10.8 *Un insieme di task con RMS [WARR91]*

Una misura dell'efficacia di un algoritmo di scheduling periodico, è se riesce a garantire, o no, che siano soddisfatte tutte le scadenze rigide: supponiamo di avere n task, ciascuno dei quali con un periodo ed un tempo d'esecuzione fissati. Allora, perché sia possibile soddisfare tutte le scadenze, deve valere la seguente disuguaglianza:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq 1 \tag{10.1}$$

La somma degli utilizzi del processore dei singoli task non può superare il valore 1, che corrisponde all'utilizzo totale del processore. L'Equazione (10.1) fornisce un limite al numero dei task che un algoritmo di scheduling perfetto può schedulare con successo; ogni algoritmo particolare può avere limite inferiore. Per RMS, si può dimostrare che vale la seguente diseguaglianza:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n\,(2^{1/n} - 1) \tag{10.2}$$

La Tabella 10.4 dà alcuni valori per questo limite superiore. Al crescere del numero dei task, il limite tende a $\ln 2 \approx 0.693$.

Come esempio, consideriamo il caso di tre task periodici, dove $U_i = C_i/T_i$:

- Task P_1 : $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- Task P_2 : $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- Task P_3 : $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$.

L'utilizzo totale di questi tre task è 0.2 + 0.267 + 0.286 = 0.753. Il limite superiore per la schedulabilità di questi tre task con RMS è:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_n}{T_n} \leq 3\,(2^{1/3} - 1) = 0.779$$

Poiché l'utilizzo totale richiesto per questi tre task è inferiore al limite superiore di RMS (0.753 < 0.779), sappiamo che, se useremo RMS, i task saranno schedulati con successo.

Si può inoltre dimostrare che il limite superiore dell'Equazione (10.1) vale anche per lo scheduling earliest-deadline; quindi, è possibile ottenere la più grande e completa utilizzazione del processore e soddisfare le scadenze di più task periodici anche con lo scheduling earliest-deadline. Tuttavia, RMS è stato ampiamente adottato nelle applicazioni industriali, [SHA91] cita le seguenti motivazioni:

1. La differenza in prestazioni, in pratica, è piccola. Il limite superiore dell'Equazione (10.2) è prudente e, in pratica, viene spesso raggiunta un'utilizzazione del 90%.
2. La maggior parte dei sistemi hard real time ha anche componenti soft real time, ad esempio alcune visualizzazioni non critiche ed autodiagnostica incorporata, che possono essere eseguite a livelli di priorità inferiore, per assorbire il tempo di processore non usato dallo scheduling RMS di task hard real time.
3. La stabilità è più facile da raggiungere con RMS. Quando un sistema non può soddisfare tutte le scadenze a causa di sovraccarico o di errori transienti, le scadenze dei task essenziali devono essere garantite, purché tale sottoinsieme di task sia schedulabile. In un approccio tipo assegnazione a priorità statica, si deve solo assicurarsi che i task essenziali abbiano

Tabella 10.4 *Valore del limite superiore di RMS*

n	$n(2^{1/n} - 1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
...	...
∞	$\ln 2 \approx 0.693$

priorità relativamente alte: questo si può fare con RMS strutturando i task essenziali in modo che essi abbiano periodi brevi, o modificando le priorità RMS per tener conto dei task essenziali. Con lo scheduling earliest-deadline, la priorità di un task periodico cambia da un periodo all'altro, e questo rende più difficile garantire che i task essenziali soddisfino le loro scadenze.

10.3 Scheduling in UNIX SVR4

L'algoritmo di scheduling usato in UNIX SVR4 è una completa revisione dell'algoritmo di scheduling usato nei sistemi UNIX precedenti. Il nuovo algoritmo è progettato per dare la più alta preferenza ai processi in tempo reale, la seconda preferenza ai processi in modo kernel e preferenza inferiore agli altri processi in modo utente, detti processi in time-sharing.

Le due maggiori modifiche implementate in SVR4 sono le seguenti:

1. L'aggiunta di uno scheduler a priorità statica prerilasciabile e l'introduzione di un insieme di 160 livelli di priorità suddivisi in tre classi di priorità.
2. L'inserimento di punti di prerilascio. Poiché il kernel base non ha prerilascio, può solo essere diviso in passi di elaborazione che devono essere eseguiti fino al termine senza interruzione. Tra i passi di elaborazione, sono state identificate zone *sicure* (*safe* places), note come punti di prerilascio, dove il kernel può interrompere in modo sicuro l'elaborazione e schedulare un processo nuovo. Una zona sicura è definita come una regione di codice dove tutte le strutture di dati del kernel sono aggiornate e consistenti, oppure bloccate da un semaforo.

La Figura 10.9 illustra i 160 livelli definiti in SVR4. Ogni processo, per definizione, appartiene ad una delle tre classi di priorità, e gli viene assegnato un livello di priorità entro la classe. Le classi sono le seguenti:

- **Tempo Reale (159-100):** Si garantisce che i processi a questi livelli di priorità sono selezionati per l'esecuzione prima di ogni processo del kernel o time sharing, inoltre, i processi in tempo reale possono far uso di punti di prerilascio per avere prelazione sui processi del kernel e sui processi utente.
- **Kernel (99-60):** Si garantisce che i processi che appartengono a questi livelli di priorità sono selezionati per l'esecuzione prima di ogni processo time sharing, ma devono cedere il processore ai processi in tempo reale.
- **Time Sharing (59-0):** I processi a priorità inferiore, destinati alle applicazioni utente diverse dalle applicazioni in tempo reale.

La Figura 10.10 indica come è implementato lo scheduling in SVR4. Una coda di allocazione è associata ad ogni livello di priorità ed i processi ad un dato livello di priorità sono eseguiti con sistema round-robin. Un vettore di bitmap, dqactmap, contiene un bit per ogni livello di priorità;

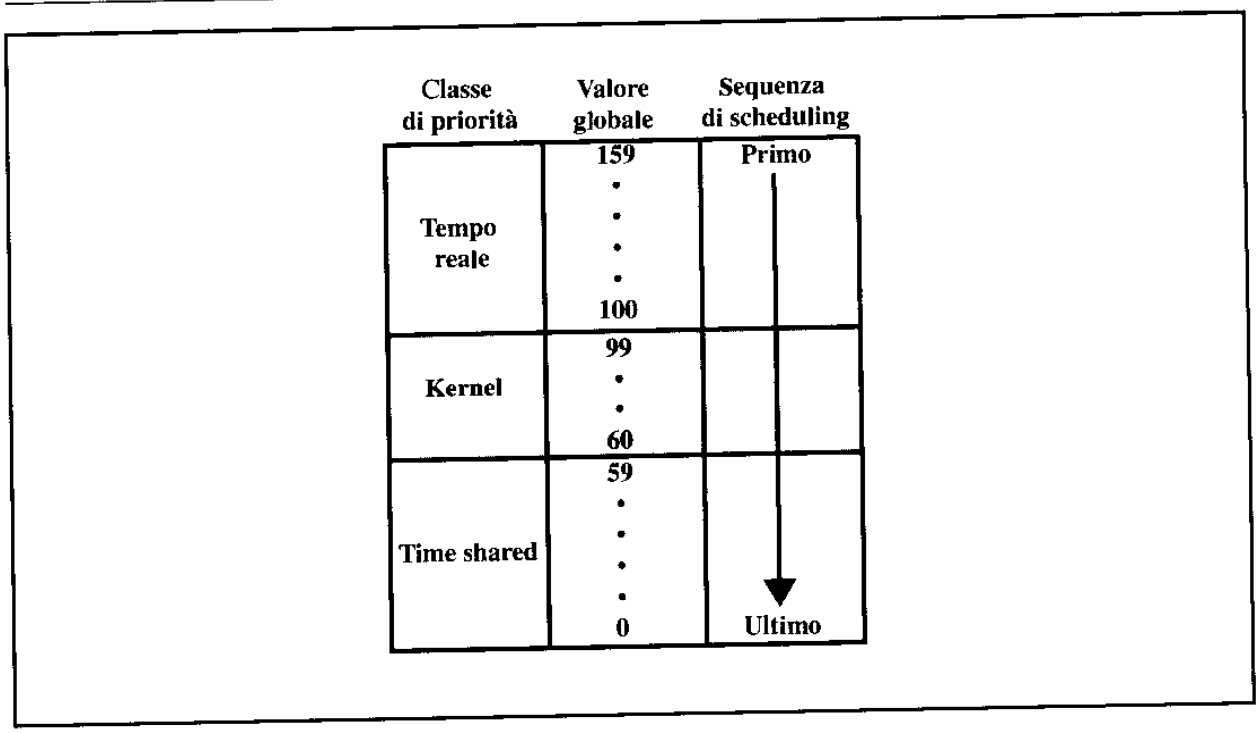


Figura 10.9 *Classi di priorità SVR4*

il bit vale 1 per ogni livello di priorità con una coda non vuota. Ogni volta che un processo in esecuzione lascia lo stato Running, per un blocco, al termine del quanto di tempo, oppure viene prerilasciato, l'allocatore controlla dqactmap ed alloca un processo Ready, preso dalla coda a più alta priorità non vuota. Inoltre, ogni volta che si raggiunge un punto di prerilascio, il kernel controlla un flag chiamato kprunrun. Se vale 1, questo indica che almeno un processo in tempo reale è nello stato Ready, ed il kernel prerilascia il processo corrente, se ha priorità inferiore rispetto al processo in tempo reale Ready con la più alta priorità.

Entro la classe time sharing, la priorità di un processo è variabile: lo scheduler riduce la priorità di un processo ogni volta che esaurisce il suo quanto di tempo, ed innalza la sua priorità se si blocca su un evento od una risorsa. Il quanto di tempo allocato ad un processo in tempo reale dipende dalla sua priorità, e va dai 100 ms per priorità 0, ai 10 ms per priorità 59. Ogni processo in tempo reale ha una priorità ed un quanto di tempo fissati.

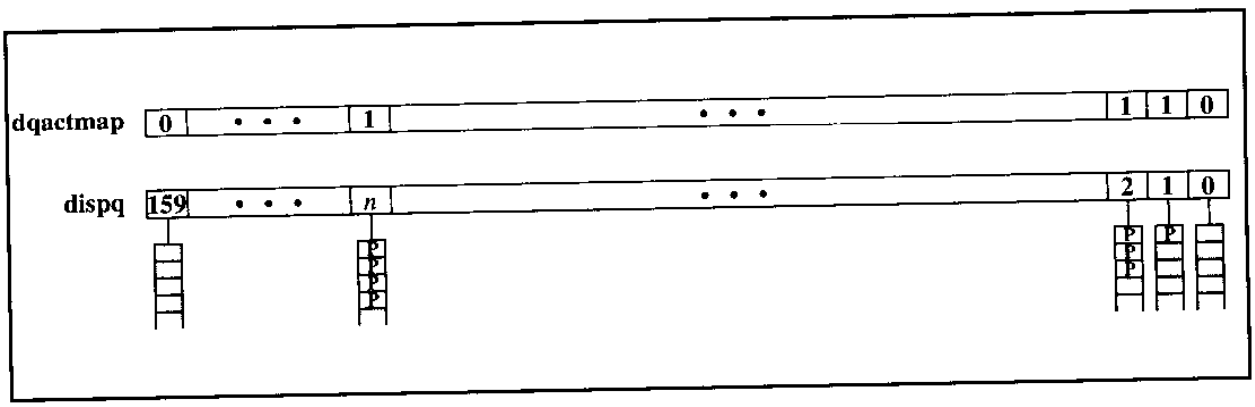


Figura 10.10 *Code di smistamento SVR4*

10.4 Scheduling in Windows NT

Windows NT è progettato in modo da essere il più pronto possibile ai bisogni di un singolo utente in un ambiente altamente interattivo, o nel ruolo di server. Windows NT implementa uno scheduler con prerilascio con un sistema flessibile di livelli di priorità, che contiene lo scheduling round-robin all'interno di ogni livello e, per alcuni livelli, variazioni di priorità dinamiche in base all'attività corrente dei thread.

Priorità di un processo e di un thread

Le priorità in Windows NT sono organizzate dentro due bande, o classi: a tempo reale e variabile. Ciascuna di queste bande è formata da 16 livelli di priorità. I thread che richiedono attenzione immediata sono nella classe in tempo reale, che contiene funzioni quali, ad esempio, le comunicazioni ed i task in tempo reale.

In complesso, poiché NT fa uso di uno scheduler con prerilascio guidato dalla priorità, i thread con priorità in tempo reale hanno la precedenza su tutti gli altri thread. Su un monoprocessore, quando diventa Ready un thread la cui priorità è più alta del thread correntemente in esecuzione, il thread a priorità più bassa viene prerilasciato, ed il processore viene dato al thread con priorità più alta.

Le priorità sono gestite in modo diverso nelle due classi (Figura 10.11). Nella classe a priorità in tempo reale, tutti i thread hanno una priorità fissata che non cambia mai; e tutti i thread attivi con un dato livello di priorità, sono in una coda round-robin. Nella classe a priorità variabile, la priorità del thread inizia ad un qualche valore iniziale assegnato, ma poi può cambiare, in su o in giù, durante il tempo di vita del thread. Quindi, c'è una coda FIFO ad ogni livello di priorità, ma un processo può migrare in una delle altre code entro la classe di priorità variabile. Tuttavia, un thread a livello di priorità 15 non può essere promosso a livello 16 e neppure ad altri livelli nella classe tempo reale.

La priorità iniziale di un thread nella classe di priorità variabile viene determinata da due quantità: priorità base del processo e priorità base del thread. Uno degli attributi di un oggetto processo è la *priorità base del processo* (*process base priority*), che può assumere un qualsiasi valore da 0 a 15. Ogni oggetto thread associato ad un oggetto processo ha un attributo *priorità base del thread* (*thread base priority*), che indica la priorità base di un thread relativamente a quella del processo: la priorità base di un thread può essere uguale a quella del suo processo, ma anche sopra o sotto quella del processo, di uno o due livelli. Così, per esempio, se un processo ha una priorità base di 4, ed uno dei suoi thread ha priorità base -1, allora la priorità iniziale di quel thread è 3.

Una volta che un thread è stato attivato nella classe di priorità variabile, la sua priorità corrente, detta priorità dinamica del thread, può fluttuare entro limiti dati: la priorità dinamica non può mai scendere sotto il livello minimo della priorità di base del thread e non può mai superare 15. La Figura 10.12 mostra un esempio: l'oggetto processo ha un attributo di priorità base pari a 4. Ogni oggetto thread associato con quest'oggetto processo deve avere una priorità iniziale tra 2 e 6; la priorità dinamica per tale thread può fluttuare fra 2 e 15 per i motivi che seguono: se un thread viene interrotto perché ha esaurito il suo quanto di tempo corrente, NT Executive diminu-

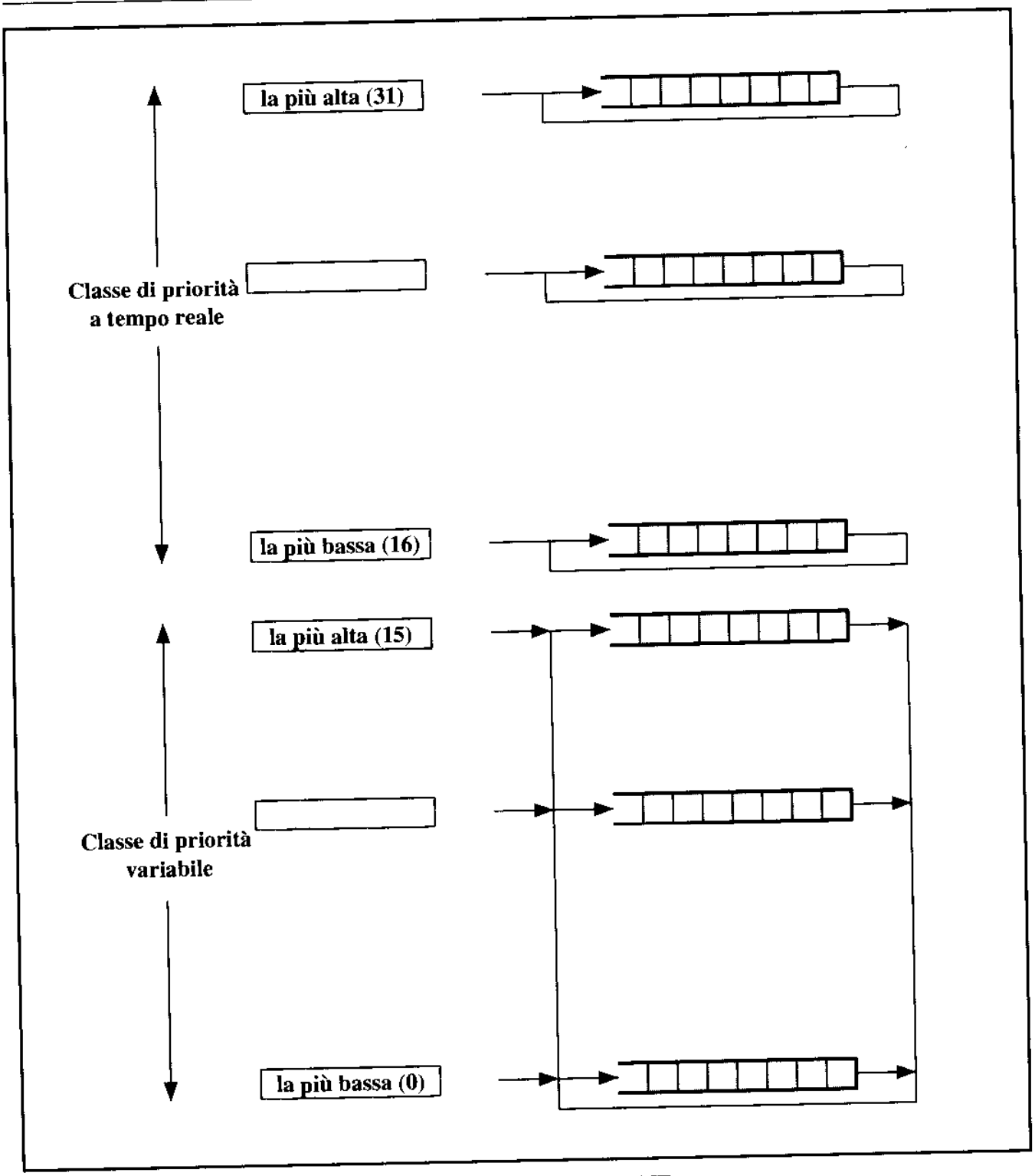


Figura 10.11 *Priorità di allocazione dei thread in Windows NT*

isce la sua priorità, mentre se viene interrotto per aspettare un evento di I/O, NT Executive aumenta la sua priorità. Quindi i thread processor bound tendono verso le priorità inferiori, e i thread I/O bound tendono verso le priorità superiori. Nel caso di thread I/O bound, NT Executive aumenta la priorità più per attese interattive (ad es. attesa della tastiera o del video) che per altri tipi di I/O (ad es. I/O da disco), quindi, i thread interattivi tendono ad avere le priorità più alte all'interno della classe di priorità variabile.

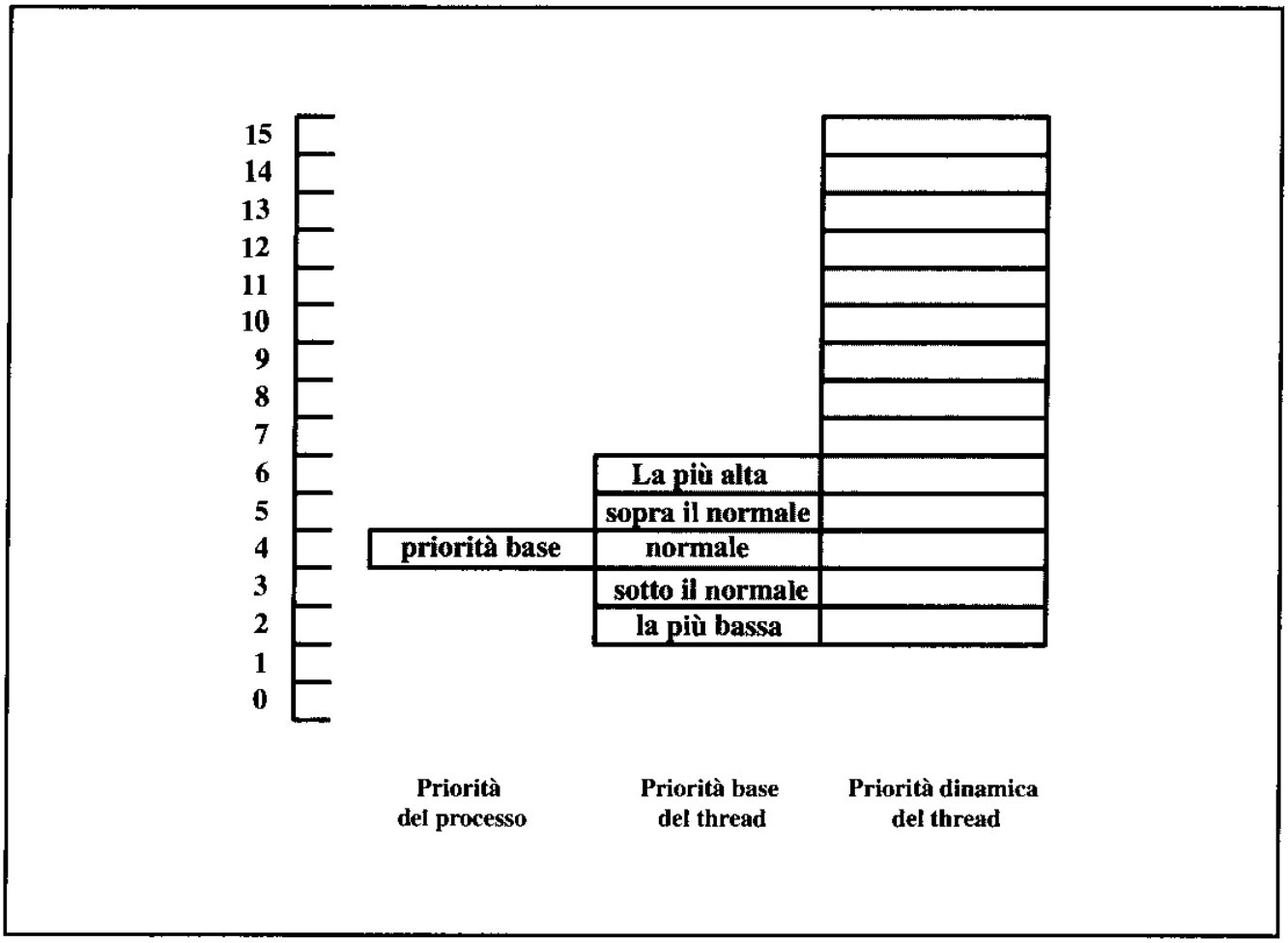


Figura 10.12 *Esempio di relazioni di priorità in Windows NT*

Scheduling multiprocessore

Quando NT è in esecuzione su un processore singolo, il thread con la priorità più alta è sempre attivo, a meno che non stia aspettando un evento. Se c'è più di un thread che ha la priorità più alta, allora il processore è condiviso, round-robin, fra tutti i thread con quel livello di priorità. In un sistema multiprocessore con N processori, gli (N - 1) thread con la priorità più alta sono sempre attivi, in esecuzione esclusivamente sugli (N - 1) processori extra. I rimanenti thread, a priorità inferiore, condividono il singolo processore rimanente. Per esempio, se ci sono tre processori, i due thread con la priorità più alta sono in esecuzione su due processori, mentre tutti i rimanenti thread vengono eseguiti sul processore rimanente.

La disciplina precedente è influenzata dall'attributo di affinità dei processori di un thread. Se un thread è Ready, ma i soli processori disponibili non sono nel suo insieme di affinità dei processori, allora il thread è costretto ad aspettare, e NT Executive schedula il successivo thread disponibile.

10.5 Sommario

In un multiprocessore strettamente accoppiato, processori multipli hanno accesso alla stessa memoria principale; in questa configurazione, la struttura di scheduling è qualcosa di più complesso. Per esempio un dato processo può essere assegnato allo stesso processore per l'intera durata del suo tempo di vita, o inviato a un processore qualsiasi ogni volta che entra nello stato Running. Studi sulle prestazioni suggeriscono che le differenze tra i vari algoritmi di scheduling sono meno significative in un sistema multiprocessore.

Un processo o un task in tempo reale viene eseguito in relazione con processi, funzioni o insiemi di eventi esterni al sistema, e deve soddisfare uno o più scadenze per interagire efficacemente e correttamente con l'ambiente esterno. Un sistema operativo in tempo reale deve organizzare i processi in tempo reale: in questo contesto, i tradizionali criteri per scegliere un algoritmo di scheduling, non valgono più. Il fattore chiave è, invece, quello di soddisfare le scadenze: in queste condizioni si usano algoritmi che fanno pesantemente uso di prerilascio e reagiscono alle scadenze relative.

10.6 Letture raccomandate

[WEND89] è una discussione interessante di approcci allo scheduling multiprocessore. Le seguenti collezioni di articoli recenti contengono tutte articoli importanti sui sistemi operativi in tempo reale e sullo scheduling: [KRIS94], [STAN93], [LEE93] e [TILB91]. [ZEAD97] analizza le prestazioni dello scheduler in tempo reale di SVR4.

KRIS94 Krishna, C., e Lee, Y. eds. "Special Issue on Real Time Systems." *Proceedings of the IEEE*, Gennaio 1994.

LEE93 Lee, Y., e Krishna, C., eds. *Readings in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

STAN93 Stankovich, J., e Ramamritham, K., eds. *Advances in Real-Time Systems*. Los Alamitos CA: IEEE Computer Society Press, 1993.

TILB91 Tilborg, A., e Koob, G., eds. *Foundations of real time Computing: Scheduling and Resource Management*. Boston: Kluwer Academic Publishers, 1991.

WEND89 Wendorf, J., Wendorf, R.; e Tokuda, H. "Scheduling Operative System Processing on Small-Scale Microprocessors." *Proceedings, 22nd Annual Hawaii International Conference on System Science*, Gennaio 1989

ZEAD97 Zeadally, S. "An Evaluation of the real time Performance of SVR4.0 and SVR4.2." *Operating Systems Review*, Gennaio 1977.

10.7 Problemi

10.1 Consideriamo un insieme di tre task periodici con i profili d'esecuzione della Tabella 10.5. Sviluppare diagrammi di scheduling simili a quelli di Figura 10.5 per questo insieme di task.

Tabella 10.5 *Profilo d'esecuzione per il Problema 10.1*

Processo	Tempo d'arrivo	Tempo d'esecuzione	Scadenza di completamento
A(1)	0	10	20
A(2)	20	10	40
...	...	...	...
B(1)	0	10	50
B(2)	50	10	100
...	...	...	...
C(1)	0	15	50
C(2)	50	15	100
...	...	...	...

10.2 Consideriamo un insieme di cinque task aperiodici con i profili di esecuzione della Tabella 10.6. Sviluppare diagrammi di scheduling simili a quelli di Figura 10.6 per questo insieme di task.

Tabella 10.6 *Profilo di esecuzione per il Problema 10.2*

Processo	Tempo d'arrivo	Tempo d'esecuzione	Scadenza di inizio
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

10.3 Questo problema dimostra che, sebbene l'Equazione (10.2) per lo scheduling a frequenza monotona sia una condizione sufficiente per uno scheduling con successo, non è una condizione necessaria [in altre parole, qualche volta è possibile che uno scheduling abbia successo anche se l'Equazione (10.2) non è soddisfatta].

a. Si consideri un insieme di task con i seguenti task periodici indipendenti:

- Task P_1 : $C_1 = 20$; $T_1 = 100$
- Task P_2 : $C_2 = 30$; $T_2 = 145$

Questi task possono essere schedulati con successo usando lo scheduling a frequenza monotona?

b. Ora si aggiunga all'insieme il seguente task:

- Task P_3 : $C_3 = 68$; $T_3 = 150$

L'Equazione (10.2) è soddisfatta?

c. Si supponga che la prima istanza dei tre task precedenti arrivi al tempo $t = 0$, e che la prima scadenza per ogni task sia la seguente:

$D_1 = 100$; $D_2 = 145$; $D_3 = 150$

Usando lo scheduling a frequenza monotona, saranno soddisfatti tutte e 3 le scadenze? Cosa potete dire sulle scadenze, per le ripetizioni future di ciascun task?

INPUT/OUTPUT E FILE

Parte 5

Forse la parte più confusa della progettazione di un sistema operativo riguarda l'I/O e il sistema di gestione dei file. Dal punto di vista dell'I/O, la parola chiave è prestazioni, poiché il servizio dell'I/O è effettivamente il terreno di battaglia delle prestazioni. Analizzando le operazioni interne di un sistema di elaborazione, vediamo che le velocità dei processori continuano ad aumentare e, se un singolo processore non è ancora sufficientemente veloce, configurazioni SMP forniscono processori multipli che accelerano il lavoro. Gli accessi alla memoria interna si stanno a loro volta velocizzando, anche se non alla pari dei processori; in ogni modo, con un uso sapiente di uno, due o anche più livelli di cache interna, l'accesso alla memoria centrale sta tenendo testa alle velocità dei processori. Invece l'I/O rimane una continua sfida alle prestazioni, in particolare nel caso del disco.

Anche nel caso dei file system, le prestazioni sono un aspetto importante, inoltre entrano in gioco anche altri requisiti di progettazione quali affidabilità e sicurezza. Dal punto di vista dell'utente, il file system è forse l'aspetto più importante del sistema operativo: l'utente vuole accessi rapidi ai file, ma vuole anche che i file non vengano corrotti e che gli accessi ai file siano autorizzati.

Capitolo 11
Gestione dell'I/O e schedulazione del disco

Il Capitolo 11 comincia con una panoramica dell' I/O sui dispositivi di memorizzazione e delle funzioni di I/O all'interno del sistema operativo; a questo segue una discussione su varie strategie di bufferizzazione utilizzate per migliorare le prestazioni. Il resto del capitolo è dedicato all'I/O di disco. Vedremo i modi in cui richieste multiple di disco possono essere schedulate, per sfruttare le caratteristiche fisiche dell'accesso

a disco e migliorare il tempo di risposta. In seguito esamineremo l'uso di un array di dischi per migliorare prestazioni e affidabilità. Infine, discuteremo della cache di disco.

Capitolo 12
Gestione dei file

Il Capitolo 12 fornisce una panoramica dei vari tipi di organizzazioni di file, ed esamina gli aspetti del sistema operativo relativi alla gestione dei file e all'accesso ai file.

CAPITOLO 11

GESTIONE DELL'I/O E SCHEDULAZIONE DEL DISCO

Forse l'aspetto più confuso nella progettazione dei sistemi operativi è l'input/output. Esiste un gran numero di dispositivi e di applicazioni per tali dispositivi, ed è quindi difficile sviluppare una soluzione consistente e generale.

Iniziamo questo capitolo con una breve descrizione dei dispositivi di I/O e dell'organizzazione delle funzioni di I/O. Questi argomenti, che solitamente si collocano nell'ambito delle architetture dei computer, forniscono la base per una trattazione dell'I/O dal punto di vista dei sistemi operativi.

Nella sezione successiva, esamineremo gli aspetti della progettazione di sistemi operativi, compresi gli obiettivi di progettazione e la modalità con cui le funzioni di I/O possono essere strutturate; in seguito sarà esaminata la gestione di buffer dell'I/O, uno dei servizi base di I/O forniti dal sistema operativo, che ne migliora le prestazioni complessive.

Le sezioni successive del capitolo sono dedicate all'I/O dei dischi magnetici; nei sistemi attuali questa è la forma più importante di I/O ed è la chiave delle prestazioni percepite dall'utente. Inizieremo sviluppando un modello di prestazioni dell'I/O del disco; in seguito esamineremo svariate tecniche che possono essere utilizzate per migliorare le prestazioni.

In un'appendice a questo capitolo saranno riassunte le caratteristiche dei dispositivi di memorizzazione secondaria, quali i dischi magnetici, i nastri magnetici e la memoria ottica.

11.1 Dispositivi di I/O

Come è stato accennato nel Capitolo 1, i dispositivi esterni che consentono l'I/O con i sistemi di elaborazione possono essere sommariamente divisi in tre categorie:

- **Leggibili dall'uomo:** adatti a comunicare con l'utente, ad esempio i terminali video, composti di video, tastiera, e forse altri dispositivi quali il mouse e le stampanti.
- **Leggibili dalla macchina:** adatti a comunicare con l'apparecchiatura elettronica. Esempi sono i drive di dischi e nastri, i sensori, i controller e gli attuatori.
- **Di comunicazione**: adatti a comunicare con dispositivi remoti, ad esempio, i driver di linea digitale e i modem.

Ci sono grandi differenze tra le classi e, addirittura, differenze sostanziali all'interno della stessa classe. Tra le differenze principali, distinguiamo:

- **Quantità di dati**: vi possono essere differenze di diversi ordini di grandezza tra le velocità di trasferimento dei dati; la Tabella 1.1 fornisce qualche esempio.
- **Applicazioni:** l'uso di un particolare dispositivo influenza il software e le politiche dei sistemi operativi e delle utilità di supporto. Per esempio, usare un disco per i file richiede il software di gestione dei file, mentre l'uso di un disco per memorizzare pagine in uno schema

Tabella 11.1 *Esempi di dispositivi di I/O classificati per comportamento, partner e frequenza di dati [HENN90]*

Dispositivo	Scopo	Partner	Frequenza di dati (kilobyte/s)
Tastiera	Input	Umano	0.01
Mouse	Input	Umano	0.02
Input vocale	Input	Umano	0.02
Scanner	Input	Umano	200
Output vocale	Output	Umano	0.6
Stampante a linee	Output	Umano	1
Stampante laser	Output	Umano	100
Video grafici	Output	Umano	30 000
CPU verso il buffer di frame	Output	Umano	200
Rete – terminali	Input o output	Macchina	0.05
Rete – LAN	Input o output	Macchina	200
Disco ottico	Immagazzinamento	Macchina	500
Nastro magnetico	Immagazzinamento	Macchina	2000
Disco magnetico	Immagazzinamento	Macchina	2000

di memoria virtuale dipende dall'hardware e dal software per la memoria virtuale; inoltre, queste applicazioni influenzano gli algoritmi di schedulazione del disco (discussi nel seguito di questo capitolo). Come altro esempio, un terminale può essere usato da un utente normale, o da un amministratore di sistema; tali utilizzi implicano diversi livelli di privilegio e, forse, diverse priorità nel sistema operativo.

- **Complessità di controllo:** una stampante richiede un'interfaccia di controllo relativamente semplice; un disco è molto più complicato. L'effetto di queste differenze sul sistema operativo è in parte filtrato dalla complessità del modulo di I/O che controlla il dispositivo, come sarà approfondito nella prossima sezione.
- **Unità di trasferimento:** i dati possono essere trasferiti come un flusso di byte o di caratteri (ad esempio nell'I/O del terminale) o in blocchi più grandi (ad esempio nell'I/O del disco).
- **Rappresentazione dei dati:** diversi dispositivi usano differenti schemi di codifica dei dati, compresi differenti codici dei caratteri e convenzioni di parità.
- **Condizioni di errore**: la natura degli errori, i modi in cui sono segnalati, le loro conseguenze e la gamma di risposte, differiscono ampiamente da un dispositivo ad un altro.

A causa di queste differenze è difficile ottenere un approccio uniforme e consistente all'I/O, sia dal punto di vista del sistema operativo, sia dal punto di vista dei processi utente.

11.2 Organizzazione delle funzioni di I/O

Nella Sezione 1.7 sono state elencate tre tecniche per effettuare I/O:

- **I/O programmato**: il processore genera un comando di I/O verso un modulo di I/O per conto di un processo; a questo punto il processo rimane in attesa attiva aspettando, prima di procedere, che l'operazione sia completata.
- **I/O interrupt-driven** *(guidato dalle interruzioni)*: il processore genera un comando di I/O per conto di un processo, continua ad eseguire le istruzioni seguenti, e viene interrotto dal modulo di I/O quando quest'ultimo ha completato il suo lavoro. Le istruzioni successive possono appartenere allo stesso processo, se non è necessario che esso aspetti il completamento dell'I/O, altrimenti, il processo è sospeso in attesa dell'interrupt e nel frattempo viene eseguito un altro lavoro.
- **DMA** (*accesso diretto in memoria*, Direct Memory Access): un modulo di DMA controlla lo scambio di dati tra la memoria principale e un modulo di I/O; il processore invia una richiesta di trasferimento di un blocco di dati al modulo di DMA, e viene interrotto solo dopo che l'intero blocco è stato trasferito.

La Tabella 11.2 indica la relazione tra queste tre tecniche. Nella maggior parte di sistemi di elaborazione, il DMA è la forma di trasferimento più importante, che deve essere supportata dal sistema operativo.

Tabella 11.2 *Tecniche di I/O*

	Senza interrupt	Con interrupt
Trasferimento da I/O a memoria tramite processore	I/O programmato	I/O guidato da interrupt
Trasferimento diretto da I/O a memoria		DMA (accesso diretto in memoria)

L'evoluzione della funzione di I/O

Con l'evoluzione dei sistemi di elaborazione, le singole componenti sono diventate più complesse e sofisticate, cosa che è particolarmente evidente nella funzione di I/O. I passi dell'evoluzione possono essere schematizzati nel modo seguente:

1. Il processore controlla direttamente un dispositivo periferico, come accade in semplici dispositivi controllati da un microprocessore.
2. Vengono aggiunti un controller o un modulo di I/O; il processore usa I/O programmato senza interrupt e si distacca dai dettagli specifici dell'interfaccia con dispositivi esterni.
3. Si usa la stessa configurazione del passo 2, aggiungendo l'uso di interrupt; il processore non ha bisogno di aspettare il completamento di un'operazione di I/O, quindi l'efficienza aumenta.
4. Grazie al DMA, il modulo di I/O ha un controllo diretto sulla memoria, e può muovere un blocco di dati da o per la memoria senza chiamare in causa il processore, tranne che all'inizio e alla fine del trasferimento.
5. Il modulo di I/O è potenziato allo scopo di farlo diventare un processore separato, con un insieme di istruzioni specializzate all'I/O. La CPU dirige il processore di I/O nell'esecuzione di un programma di I/O in memoria centrale; il processore di I/O cerca ed esegue queste istruzioni senza l'intervento del processore. Questo permette al processore di specificare una sequenza di attività di I/O e di essere avvertito da un interrupt solo quando l'intera sequenza è eseguita.
6. Il modulo di I/O ha una sua memoria locale e diventa, in effetti, un computer vero e proprio: con quest'architettura, la maggior parte dei dispositivi di I/O può essere controllata con un coinvolgimento minimo da parte del processore. L'uso tipico di quest'architettura è il controllo di comunicazioni tra terminali interattivi, in cui il processore di I/O si prende cura della maggior parte dei compiti riguardanti il controllo dei terminali.

Procedendo lungo il cammino dell'evoluzione, notiamo che la funzione dell'I/O viene svolta con un sempre minore coinvolgimento del processore. Il processore centrale è via via liberato dai compiti relativi all'I/O, aumentando quindi le prestazioni. Con gli ultimi due passi (5-6), avviene un cambiamento sostanziale, con l'introduzione del concetto di un modulo di I/O capace di eseguire un programma.

Una nota relativa alla terminologia: per tutti i moduli descritti dal passo 4 al passo 6, il termine DMA (*accesso diretto in memoria*, Direct Memory Access) è appropriato, in quanto tutti questi tipi prevedono il controllo diretto della memoria centrale da parte del modulo di I/O. Inoltre, il modulo descritto nel passo 5, è spesso chiamato **canale di I/O** mentre quello del passo 6, **processore di I/O**; comunque, questi termini sono applicati ad entrambe le situazioni a seconda dell'occasione. Nell'ultima parte di questa sezione, useremo il termine *canale di I/O* riferendoci ad entrambi i tipi di moduli di I/O.

Accesso diretto in memoria

La Figura 11.1 indica, in termini generali, la logica DMA. L'unità di DMA è in grado di imitare il processore e, in pratica, di subentrare al processore nel controllo del sistema, allo scopo di trasferire dati da e verso la memoria, attraverso il bus di sistema. Tipicamente, o il modulo del DMA usa il bus solo quando il processore non ne ha bisogno, oppure deve forzare il processore a sospendere temporaneamente l'operazione. Quest'ultima tecnica è la più comune e viene chiamata furto di ciclo, in quanto l'unità di DMA, in effetti, ruba il ciclo di bus.

La tecnica DMA funziona come segue: quando il processore vuole leggere o scrivere un blocco di dati, produce un comando per il modulo di DMA, spedendo al modulo di DMA le seguenti informazioni:

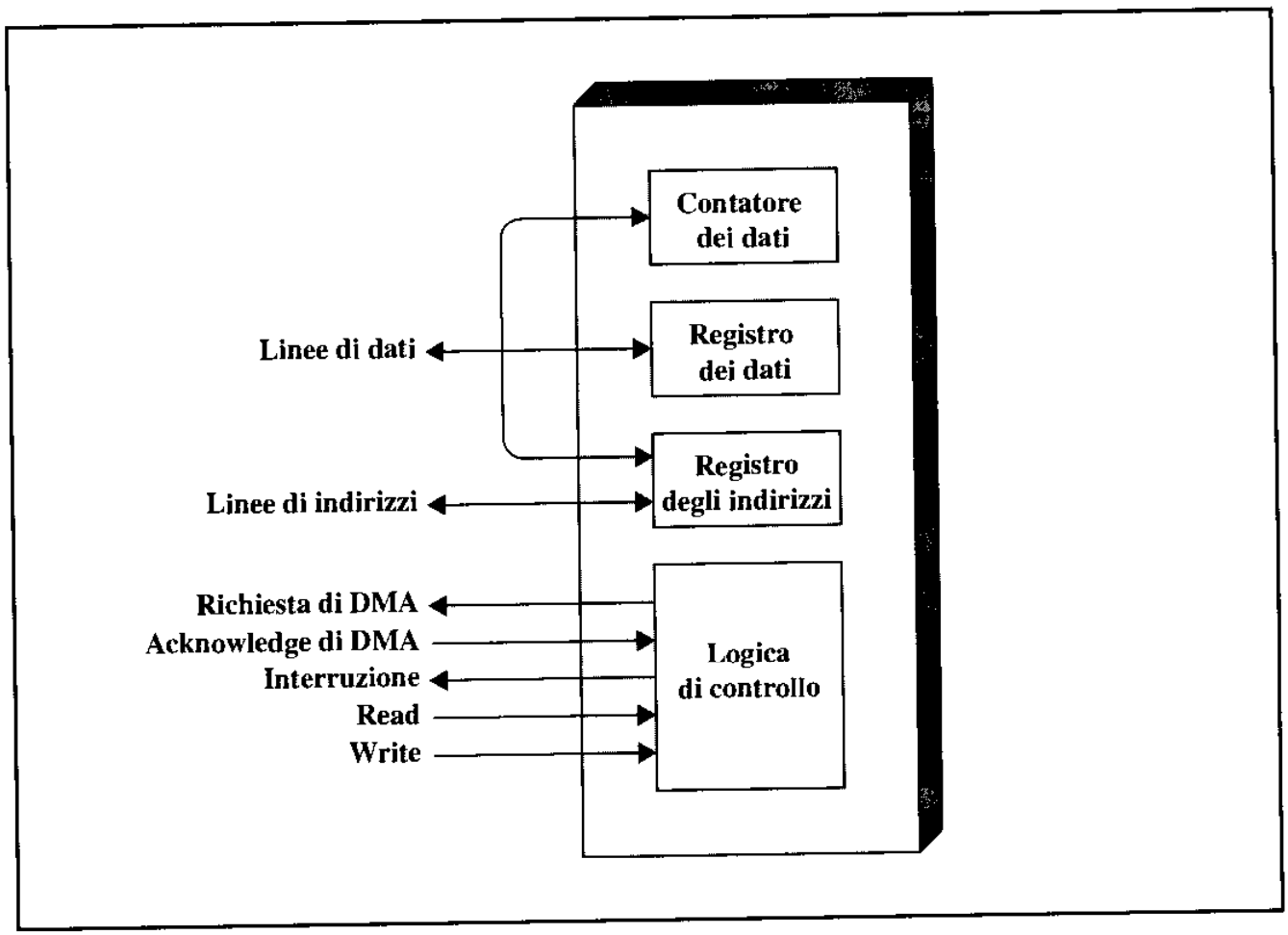


Figura 11.1 *Diagramma a blocchi di un tipico DMA*

- Se è stata richiesta una lettura o una scrittura, usando la linea di controllo di lettura o scrittura tra processore e modulo di DMA
- L'indirizzo del dispositivo di I/O coinvolto, comunicato sulle linee dati
- La posizione iniziale in memoria da cui leggere o scrivere, comunicata sulle linee dati e memorizzata dal modulo di DMA nel suo registro di indirizzi
- Il numero di parole da leggere o scrivere, ancora una volta comunicato attraverso le linee dati e memorizzato nel data count register (*registro contatore dei dati*).

A questo punto il processore può eseguire altre istruzioni, avendo delegato quest'operazione di I/O al modulo di DMA. Il modulo di DMA trasferisce l'intero blocco di dati, una parola alla volta, direttamente da o verso la memoria, senza passare attraverso il processore; quando il trasferimento è completato, il modulo di DMA invia un segnale di interrupt al processore, pertanto il processore è coinvolto soltanto all'inizio e alla fine del trasferimento (Figura 1.19c).

La Figura 11.2 mostra dove il processore può essere sospeso durante il ciclo di istruzioni. In ogni caso, il processore è sospeso immediatamente prima di avere bisogno di usare il bus; dopodiché, il DMA trasferisce una parola e restituisce il controllo al processore. Notare che questo non è un interrupt: il processore non salva un contesto per fare qualcos'altro, piuttosto, si ferma per un ciclo di bus, e l'effetto globale è che il processore esegue più lentamente. Ciò nonostante, per un trasferimento di molte parole in I/O, il DMA è molto più efficiente dell'I/O programmato o guidato da interrupt.

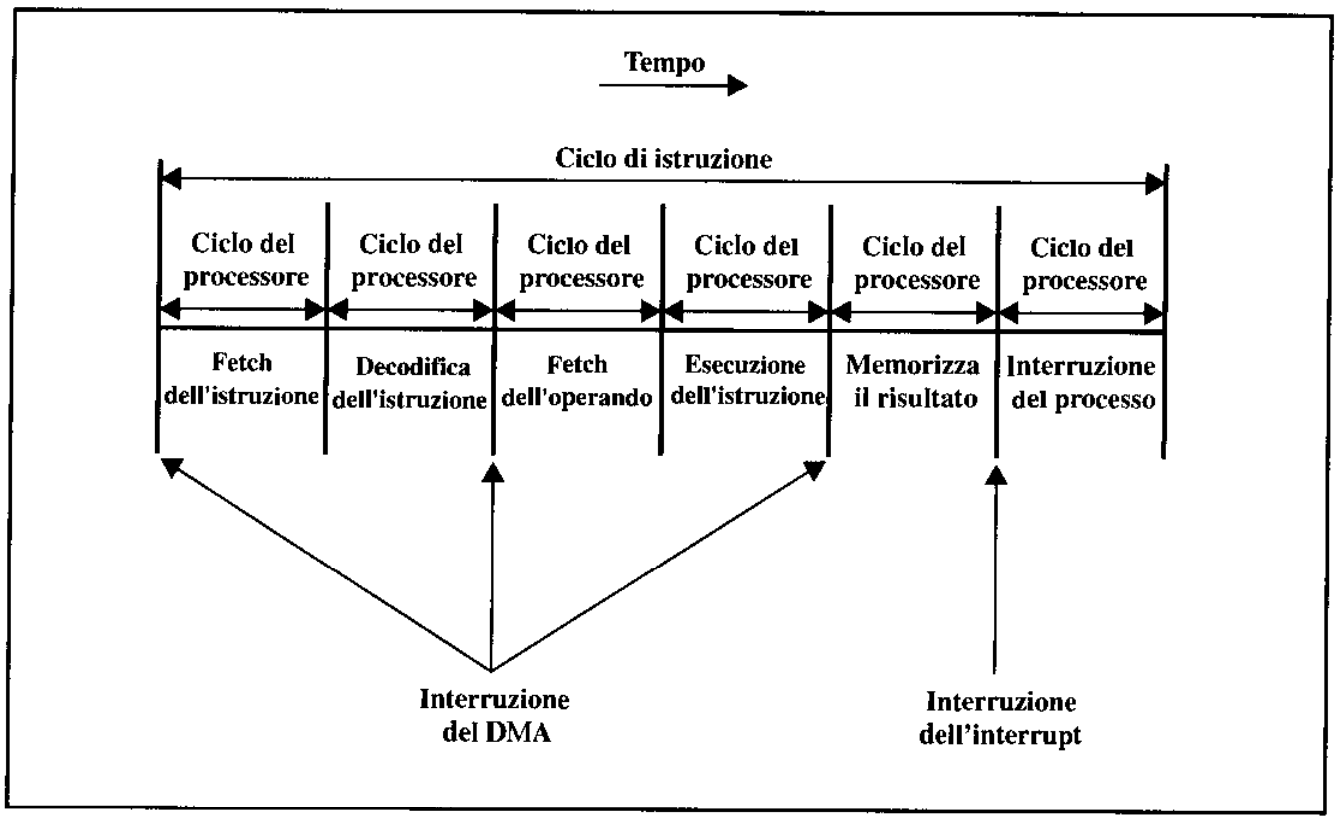


Figura 11.2 *Punti in cui il DMA e le interruzioni possono interrompere il processore durante un ciclo di istruzione*

Il meccanismo del DMA può essere configurato in vari modi, alcuni dei quali sono mostrati in Figura 11.3. Nel primo esempio, tutti i moduli condividono lo stesso bus di sistema; il modulo di DMA, agendo come surrogato del processore, usa un I/O programmato per scambiare dati tra memoria e modulo di I/O attraverso il modulo di DMA. Questa configurazione, anche se è poco costosa, è chiaramente inefficiente: come in ogni I/O programmato controllato dal processore, ogni trasferimento di parola consuma due cicli di bus.

Il numero di cicli di bus richiesti può essere tagliato drasticamente integrando il DMA e le funzioni di I/O; come indicato in Figura 11.3b, questo significa che esiste un cammino tra il modulo di DMA e uno o più moduli di I/O, esterno al bus di sistema. La logica DMA può effettivamente far parte di un modulo di I/O, o può essere un modulo separato che controlla uno

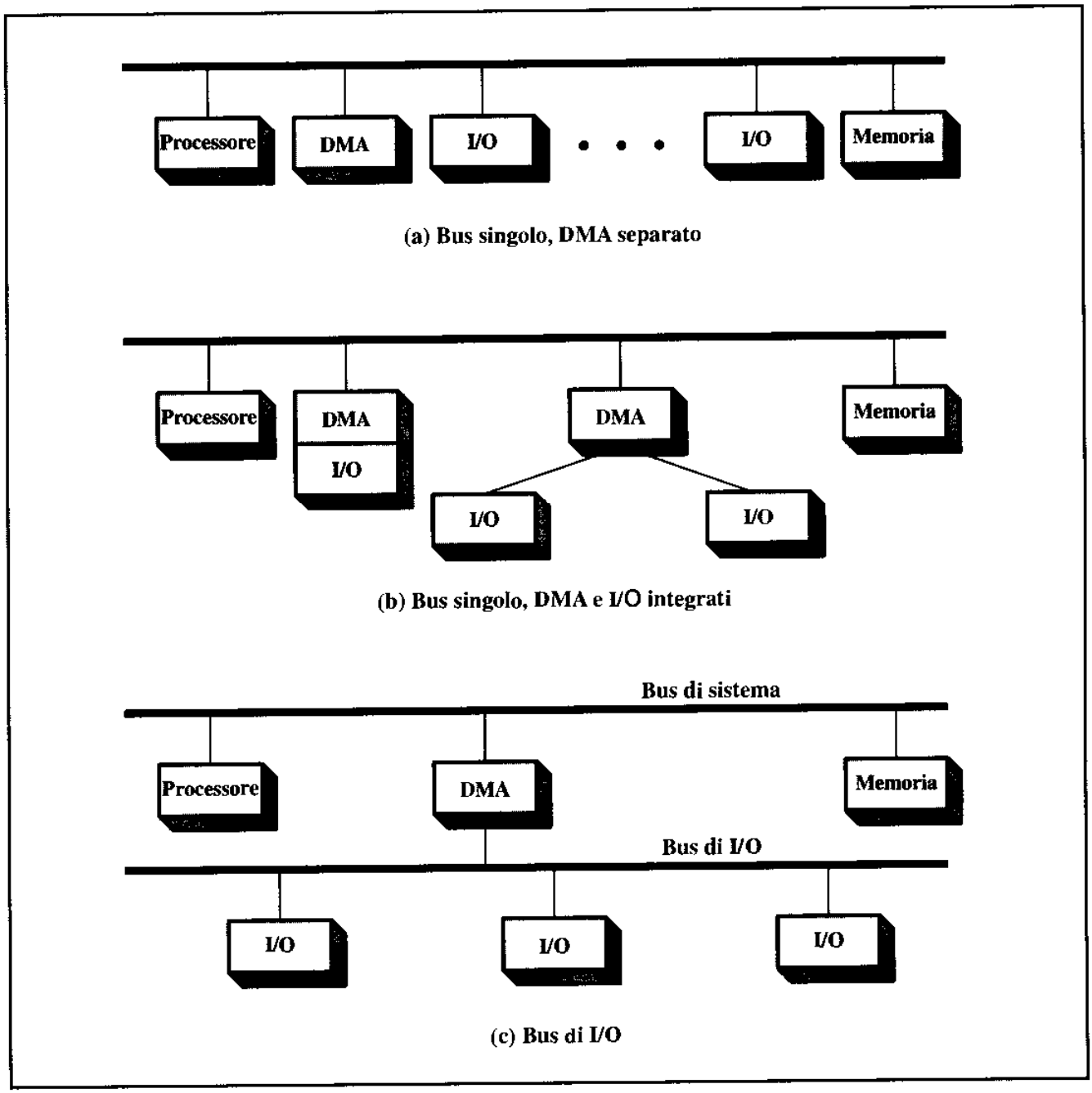


Figura 11.3 *Possibili configurazioni di DMA*

o più moduli di I/O. Questo concetto può essere esteso, connettendo moduli di I/O al modulo di DMA attraverso un bus di I/O (Figura 11.3c); in questo modo, vi è una sola interfaccia di I/O nel modulo di DMA, e la configurazione è facilmente espandibile. In tutti questi casi (figure 11.3b e 11.3c), il bus di sistema condiviso tra il modulo di DMA, il processore e la memoria centrale è usato dal modulo di DMA solo per scambiare dati con la memoria e per scambiare segnali di controllo con il processore; lo scambio di dati tra il DMA e i moduli di I/O avviene al di fuori del bus di sistema.

11.3 Progettazione di sistemi operativi

Obiettivi di progettazione

Due sono gli obiettivi sovrani nella progettazione di elementi per l'I/O: efficienza e generalità. L'**efficienza** è importante perché le operazioni di I/O formano spesso un collo di bottiglia nei sistemi di elaborazione; guardando ancora una volta la Tabella 11.1, notiamo che la maggior parte dei dispositivi di I/O sono estremamente lenti in confronto alla memoria centrale e al processore. Un modo per gestire questo problema è la multiprogrammazione, che, come abbiamo visto, permette che alcuni processi rimangano in attesa di operazioni di I/O, mentre altri processi sono in esecuzione. Comunque, anche se le macchine di oggi hanno memorie centrali di considerevoli dimensioni, succede spesso che l'I/O non stia al passo con le attività del processore. Trasferendo su disco i processi bloccati, potranno partire altri processi, che tengano il processore occupato, ma anche questo richiede operazioni di I/O. Pertanto, il più grande sforzo nella progettazione dell'I/O è stata la progettazione di schemi che ne migliorino l'efficienza. L'area che ha ricevuto maggior attenzione, grazie alla sua importanza, è l'I/O del disco; la maggior parte di questo capitolo sarà dedicata allo studio della sua efficienza.

L'altro obiettivo importante è la **generalità**. Per semplicità, e per evitare errori, si vorrebbe trattare tutti i dispositivi in modo uniforme, sia per il modo in cui i processi vedono i dispositivi di I/O, sia per il modo in cui il sistema operativo gestisce i dispositivi e le operazioni di I/O; ma le diverse caratteristiche dei dispositivi rendono difficile ottenere in pratica una vera generalità. Quello che si può fare è usare un approccio gerarchico e modulare per la progettazione della funzione di I/O, che nasconda la maggior parte dei dettagli sui dispositivi di I/O nelle procedure a più basso livello, cosicché i processi e i livelli più alti del sistema operativo vedano i dispositivi in termini di funzioni generali, quali read (leggi), write (scrivi), open (apri), close (chiudi), lock (blocca), unlock (sblocca). Discutiamo quest'approccio di seguito.

Struttura logica della funzione di I/O

Nel Capitolo 2, discutendo della struttura del sistema, abbiamo enfatizzato la natura gerarchica dei sistemi operativi moderni: la filosofia gerarchica impone di separare le funzioni del sistema operativo, a seconda della loro complessità, della loro misura di tempo caratteristica e del loro livello di astrazione, e seguendo quest'approccio si ottiene un'organizzazione del sistema operativo in una serie di livelli. Ogni livello esegue un sottoinsieme di funzioni correlate, tra

quelle richieste dal sistema operativo; esso suppone che il livello inferiore esegua funzioni più primitive, rendendogli trasparenti i dettagli di tali funzioni, ed a sua volta fornisce servizi al livello immediatamente superiore. Idealmente, i livelli dovrebbero essere definiti in modo tale che cambiamenti in un livello non richiedano cambiamenti in altri livelli, pertanto il problema è stato decomposto in tanti sottoproblemi più facili da gestire.

In genere, i livelli inferiori hanno a che fare con intervalli di tempo più brevi; alcune parti del sistema operativo possono interagire direttamente con l'hardware del computer, dove gli intervalli di tempo hanno durate dell'ordine di pochi miliardesimi di secondo. All'altro estremo dello spettro, stanno le parti del sistema operativo che comunicano con l'utente, che invia comandi con maggiore calma, ad esempio uno ogni qualche secondo: l'uso di più livelli si adatta perfettamente a questa situazione.

Applicando questa filosofia in modo specifico agli elementi dell'I/O, otteniamo il tipo di organizzazione suggerita nella Figura 11.4 (confronta con la Tabella 2.4), dove i dettagli del-

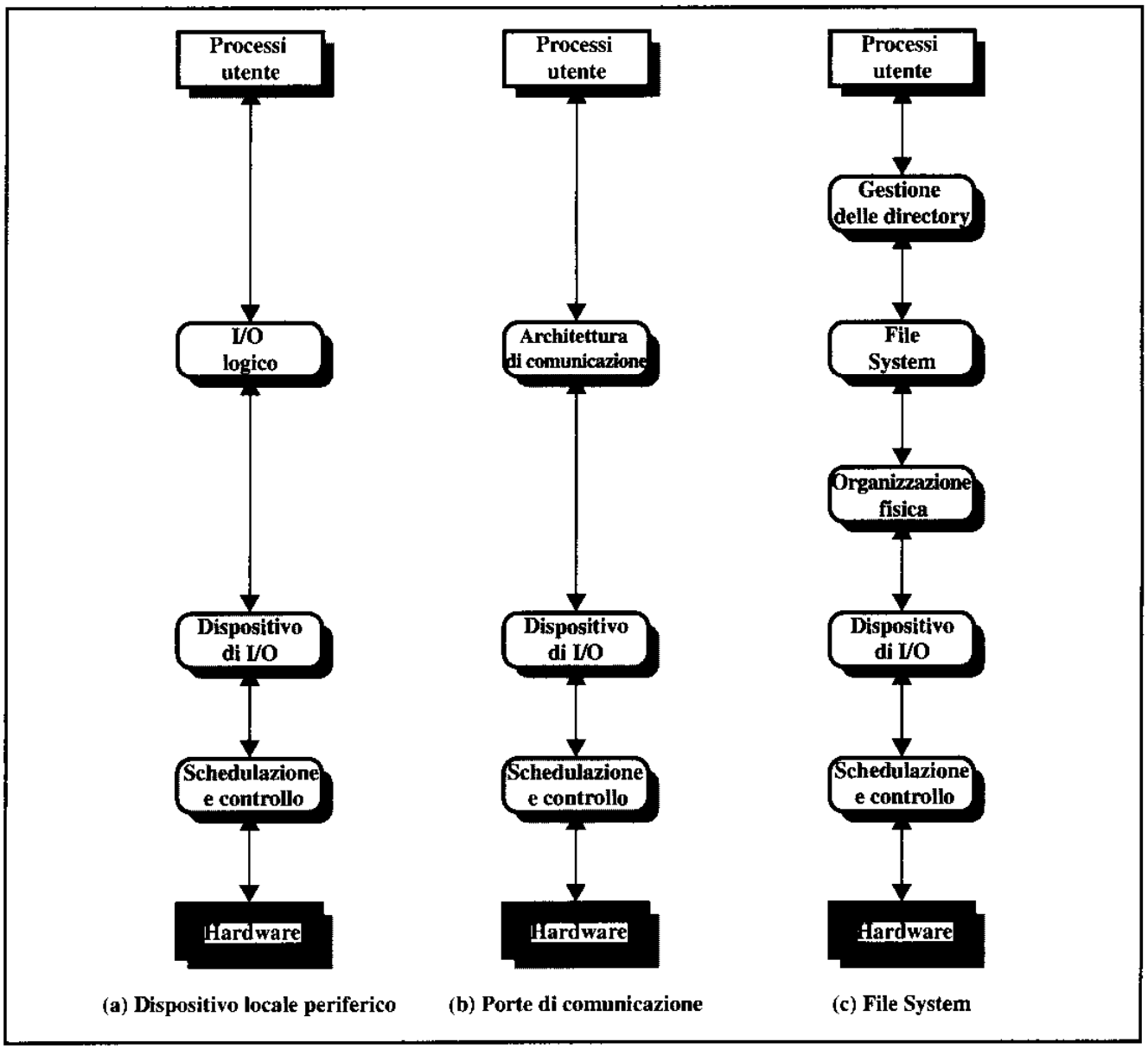


Figura 11.4 *Un modello di organizzazione dell'I/O*

l'organizzazione dipenderanno dal tipo di dispositivo e dall'applicazione. Le tre strutture logiche più importanti sono rappresentate in figura; naturalmente, un particolare sistema operativo può non adattarsi perfettamente a queste strutture, ma i principi generali sono validi e la maggior parte dei sistemi operativi gestiscono l'I/O approssimativamente in questo modo.

Consideriamo prima di tutto il più semplice caso possibile, quello di un dispositivo periferico locale che comunica in modo semplice, ad esempio con un flusso di byte o record (Figura 11.4a). I livelli coinvolti sono i seguenti:

- **I/O logico**: il modulo di I/O logico, tratta un dispositivo come una risorsa logica, senza preoccuparsi di come effettivamente controllare il dispositivo: esso riguarda la gestione delle funzioni generali di I/O per conto dei processi dell'utente, permettendo loro di trattare i dispositivi tramite un identificatore di dispositivo e semplici comandi quali open, close, read, write.
- **I/O sul dispositivo**: le operazioni richieste e i dati (caratteri nei buffer, record e così via) sono convertiti in opportune sequenze di istruzioni di I/O, comandi di canale e ordini di controllo. Per migliorare l'utilizzo si possono impiegare tecniche di gestione di buffer.
- **Schedulazione e controllo**: l'accodamento e la schedulazione delle operazioni di I/O avviene a questo livello, come anche il controllo delle operazioni; pertanto, gli interrupt sono gestiti a questo livello, in cui si salva e ripristina lo stato dell'I/O. Questo è il livello di software che effettivamente interagisce con il modulo di I/O e quindi con il dispositivo hardware.

Dal punto di vista di un dispositivo di comunicazione, la struttura dell'I/O (Figura 11.4b) appare praticamente come quella che è appena stata descritta. La principale differenza è che il modulo di I/O logico è sostituito da un'architettura di comunicazione, che può comporsi di diversi livelli; per esempio, la ben nota architettura OSI (*interconnessione di sistema aperto*, Open-System Interconnection), si compone di sette livelli. Le architetture di comunicazione saranno trattate nel Capitolo 13.

La Figura 11.4c mostra una struttura rappresentativa per la gestione dell'I/O in un dispositivo di memoria secondaria che supporta un file system. I tre livelli non ancora descritti sono i seguenti:

- **Gestione delle directory**: a questo livello, nomi simbolici di file sono convertiti in identificatori che si riferiscono al file, direttamente o indirettamente tramite un descrittore di file o una tabella di indici. Questo livello riguarda inoltre le operazioni dell'utente su directory o file, quali add (aggiungi), delete (cancella), reorganize (riorganizza).
- **File system**: questo livello tratta la struttura logica di file e le operazioni che possono essere specificate dagli utenti, quali apri, chiudi, leggi, scrivi. Anche i diritti di accesso sono gestiti a questo livello.
- **Organizzazione fisica**: come gli indirizzi di memoria virtuale, che devono essere convertiti in indirizzi fisici della memoria centrale, tenendo conto della struttura di paginazione o segmentazione, anche i riferimenti logici a file e record devono essere convertiti in indirizzi

fisici di memoria secondaria, tenendo conto della struttura fisica del file in termini di tracce e settori. Questo livello tratta inoltre l'allocazione dello spazio di memoria secondaria e dei buffer in memoria principale.

A causa dell'importanza del file system, parte di questo capitolo e del prossimo saranno dedicate ad analizzare le sue varie componenti. La discussione in questo capitolo si focalizza sul più basso dei tre livelli, mentre i due livelli superiori saranno esaminati nel Capitolo 12.

11.4 Gestione di buffer di I/O

Supponiamo che un processo utente desideri leggere blocchi di dati da un nastro, uno alla volta, e che ogni blocco sia lungo 100 byte: i dati devono essere letti in un'area dati all'interno del processo utente, dalla locazione virtuale 1000 alla 1099. Il modo più semplice sarebbe quello di eseguire un comando di I/O (qualcosa del tipo Read Block[1000,nastro]) sull'unità nastro e quindi aspettare che il dato diventi disponibile; l'attesa potrebbe essere attiva (si controlla continuamente lo stato del dispositivo) o, in modo più pratico, di tipo sospensione del processo su un interrupt.

Quest'approccio crea due problemi. Prima di tutto, il programma è sospeso ad aspettare che un I/O relativamente lento venga completato; in secondo luogo, quest'approccio all'I/O interferisce con le decisioni sui trasferimenti di processi sul disco, prese dal sistema operativo. Le locazioni virtuali dalla 1000 alla 1099 devono rimanere in memoria centrale per tutto il trasferimento del blocco, altrimenti alcuni dati potrebbero andare persi; se viene usata paginazione, almeno la pagina contenente le locazioni target deve essere mantenuta in memoria centrale. Pertanto, anche se alcune porzioni del processo possono essere trasferite su disco, è impossibile trasferire completamente l'intero processo, neanche dietro richiesta del sistema operativo. Notare inoltre che c'è il rischio di stallo di un singolo processo; se un processo invia un comando di I/O, è sospeso in attesa del risultato e viene trasferito su disco prima dell'inizio dell'operazione, allora il processo è bloccato in attesa dell'evento di I/O, mentre l'operazione di I/O è bloccata, in attesa che il processo venga riportato in memoria. Per evitare questo stallo, la memoria utente coinvolta nell'operazione di I/O deve essere bloccata in memoria centrale, immediatamente dopo che la richiesta di I/O è stata emessa, anche se l'operazione di I/O è in coda, e potrebbe non essere eseguita per un certo periodo di tempo.

Le stesse considerazioni possono essere fatte per un'operazione di output: se un blocco è in via di trasferimento da un'area di un processo utente, direttamente ad un modulo di I/O, il processo viene bloccato durante il trasferimento e quindi può non venire trasferito su disco.

Per evitare queste inefficienze, talvolta è conveniente effettuare trasferimenti di dati in input prima che sia fatta una richiesta, ed effettuare trasferimenti di output qualche tempo dopo aver fatto la richiesta. Questa tecnica è nota come gestione di buffer; in questa sezione, analizzeremo alcuni schemi di gestione di buffer, che vengono supportati dai sistemi operativi per migliorare le prestazioni del sistema.

Nel discutere i vari approcci alla gestione di buffer, talvolta è importante distinguere fra due tipi di dispositivi di I/O: dispositivi orientati ai blocchi e dispositivi orientati al flusso. I dispositivi

orientati ai blocchi (ad esempio, dischi e nastri) memorizzano l'informazione in blocchi, che solitamente sono di dimensione fissata, ed eseguono i trasferimenti un blocco alla volta; generalmente è possibile riferire i dati attraverso il loro numero di blocco. I dispositivi **orientati al flusso** (***stream***) trasferiscono dati attraverso un flusso di byte, senza struttura di blocco. Terminali, stampanti, porte di comunicazione, mouse e altri dispositivi di puntamento, nonché la maggior parte degli altri dispositivi che non siano di memoria secondaria, sono orientati a flusso.

Buffer singolo

Il più semplice tipo di supporto che un sistema operativo possa fornire è il buffer singolo (Figura 11.5b). Quando un processo utente emette una richiesta di I/O, il sistema operativo assegna all'operazione un buffer nella porzione di sistema della memoria centrale.

Per i dispositivi orientati a blocchi, lo schema di gestione di buffer singolo può essere descritto come segue: un trasferimento in input viene effettuato verso il buffer di sistema; quando è completato, il processo sposta il blocco nello spazio utente ed immediatamente richiede un altro blocco. Questa operazione viene chiamata lettura in avanti, o input anticipato, e viene eseguita supponendo che il blocco, prima o poi, verrà richiesto; per molti tipi di elaborazione questa è un'ipotesi il più delle volte ragionevole, perché solo al termine di una sequenza di elaborazione potrebbe essere letto un blocco che non è necessario.

Questo approccio generalmente migliorerà la velocità rispetto alla mancanza di gestione di buffer di sistema; il processo utente può elaborare un blocco di dati mentre si legge il blocco successivo; il sistema operativo può trasferire su disco il processo, perché l'operazione di input sta avvenendo nella memoria di sistema, invece che nella memoria del processo utente. Questa tecnica, comunque, complica la logica del sistema operativo. Il sistema operativo deve tenere traccia delle assegnazioni di buffer di sistema a processi utente, ed anche la logica dei trasferimenti su disco può essere condizionata: se l'operazione di I/O coinvolge lo stesso disco che è usato per il trasferimento del processo, non ha molto senso accodare le scritture su disco sullo stesso dispositivo. Il rilascio della memoria centrale inizia solo dopo che l'operazione di I/O è finita, e a quel punto spostare il processo su disco può essere inutile.

Simili considerazioni possono essere fatte a proposito dell'output orientato ai blocchi; quando i dati vengono trasmessi ad un dispositivo, essi sono prima di tutto copiati dallo spazio utente al buffer di sistema, dal quale essi verranno infine scritti. Il processo richiedente è quindi libero di continuare, o di essere trasferito su disco, a seconda di quanto necessario.

[KNUT97] suggerisce un confronto di prestazioni brutale, ma informativo, tra l'uso di un buffer singolo e di nessun buffer. Supponiamo che T sia il tempo richiesto per prendere in input un blocco, e che C sia il tempo di calcolo tra due richieste di input. Senza buffer, il tempo di esecuzione per blocco è essenzialmente $T+C$, con un buffer singolo è $max[C,T]+M$, dove M è il tempo richiesto per muovere i dati dal buffer di sistema alla memoria utente; nella maggior parte dei casi, quest'ultima quantità è decisamente minore della precedente.

Nel caso di I/O orientato al flusso, lo schema di buffer singolo può essere utilizzato una linea alla volta, o un byte alla volta. L'operazione una linea alla volta è opportuna per i terminali in modo di scroll (talvolta detti computer muti); con questo tipo di terminali, l'input dell'utente avviene una linea alla volta, le linee sono divise da invii, che segnalano la fine di una linea, e l'output al terminale avviene analogamente, una linea alla volta. Una stampante a linee è un

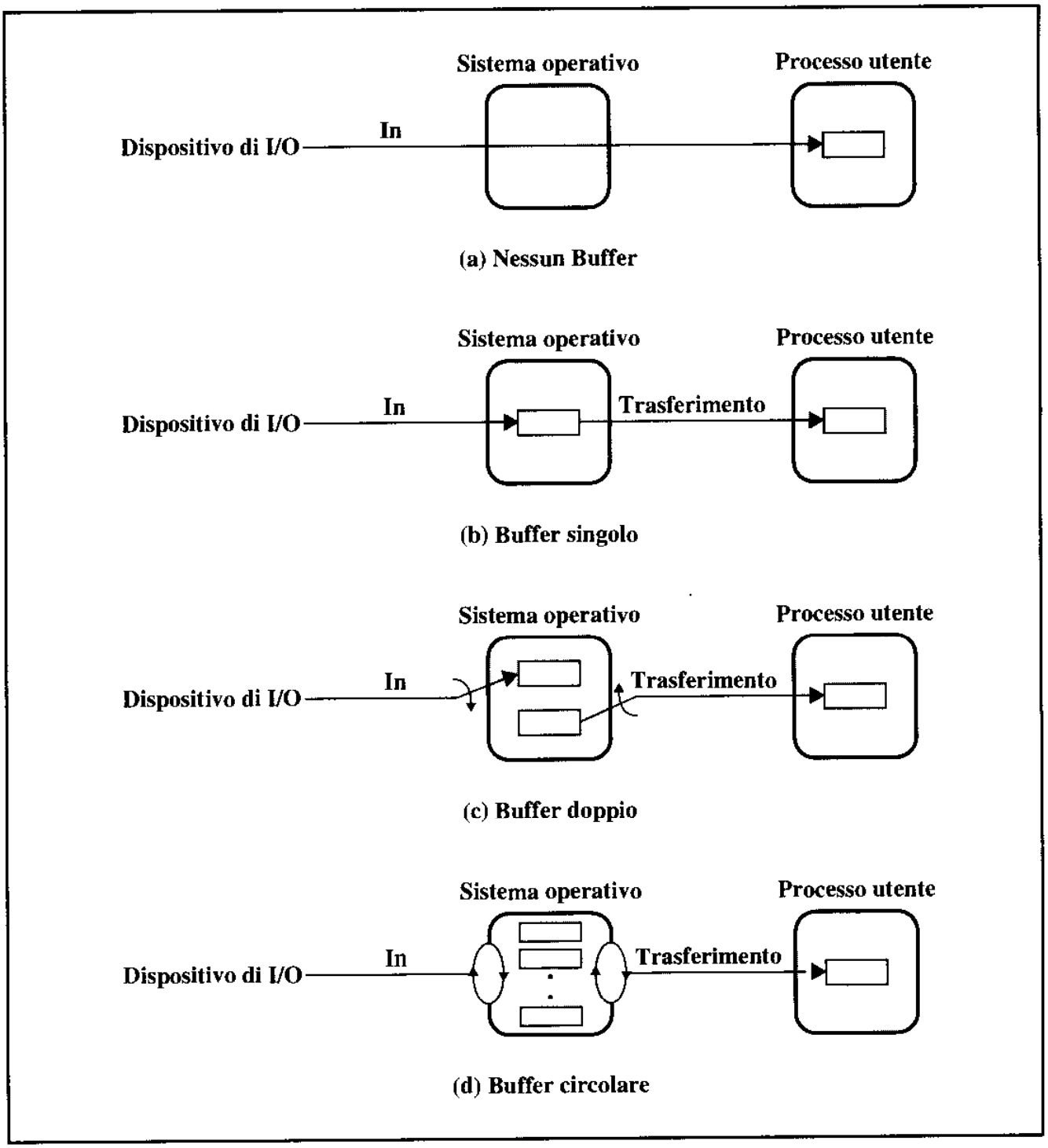


Figura 11.5 *Schemi di gestione di buffer dell'I/O (in input)*

altro esempio di questo tipo di dispositivi; le operazioni sono effettuate un byte alla volta anche sui terminali in modo form (formulario), dove ogni tasto battuto è significativo, e per molte altre periferiche, quali i sensori e i controller.

Nel caso di I/O una linea alla volta, il buffer può essere usato per contenere una singola linea; il processo utente è sospeso durante l'input, in attesa dell'arrivo dell'intera linea. Per l'output, il processo utente può mettere una linea dell'output nel buffer e continuare l'elaborazione; non ha bisogno di venire sospeso, a meno che non abbia una seconda linea di output da spedire prima

che il buffer sia stato svuotato dalla prima operazione di output. Nel caso di I/O un byte alla volta, l'interazione tra sistema operativo e processo utente segue il modello produttore/consumatore discusso nel Capitolo 5.

Buffer doppio

Un miglioramento rispetto al buffer singolo può essere ottenuto assegnando due buffer di sistema all'operazione (Figura 11.5c); in questo caso un processo trasferisce dati verso (o da) un buffer, mentre il sistema operativo svuota (o riempie) l'altro. Questa tecnica è nota col nome di **gestione di buffer doppia** o **swap di buffer**.

Nel caso di trasferimento orientato ai blocchi, possiamo grossolanamente stimare il tempo di esecuzione come il max[C,T], è quindi possibile mantenere il dispositivo orientato ai blocchi a una velocità massima, se $C<T$; d'altro canto, se $C>T$, il buffer doppio assicura che il processo non debba aspettare l'I/O. In entrambi i casi, si ottiene un miglioramento al prezzo di un aumento di complessità.

Nel caso di trasferimento orientato al flusso, ci troviamo ancora di fronte le due possibili opzioni; se l'I/O avviene una linea alla volta, il processo utente non ha bisogno di essere sospeso per l'input o l'output, a meno che il processo non vada più veloce del tempo necessario a riempire/svuotare i due buffer. Se invece l'operazione avviene un byte alla volta, il buffer doppio non offre alcun particolare vantaggio rispetto ad un buffer singolo di lunghezza doppia. In entrambi i casi viene seguito il modello del produttore/consumatore.

Buffer circolare

Uno schema di buffer doppio dovrebbe regolarizzare il flusso di dati tra un dispositivo di I/O e un processo; se le prestazioni di un particolare processo sono la nostra principale preoccupazione, vorremmo che l'operazione di I/O fosse in grado di rimanere in pari con il processo. Il buffer doppio può essere inadeguato se il processo richiede velocità mentre compie molti I/O. In questo caso il problema può venire alleviato usando più di due buffer.

Se si usano più di due buffer, il loro insieme viene chiamato buffer circolare (Figura 11.5d), dove ogni singolo buffer è un'unità del buffer circolare; si tratta semplicemente di un buffer limitato produttore/consumatore come il modello studiato nel Capitolo 5.

L'utilità della gestione di buffer

La gestione di buffer è una tecnica che regolarizza i picchi nelle richieste di I/O, ma nessuna quantità di buffer sarà sufficiente a permettere ad un dispositivo di I/O di tenere sempre il passo con un processo, se la domanda media di I/O del processo è maggiore di quella che il dispositivo può servire; anche con buffer multipli, tutti i buffer finiranno per riempirsi e il processo dovrà attendere, dopo aver elaborato ogni blocco di dati. Comunque, in un ambiente di multiprogrammazione, dove c'è una varietà di attività di I/O e una varietà di attività di processo da servire, il buffer è un mezzo che può migliorare l'efficienza del sistema operativo e le prestazioni dei singoli processi.

11.5 Schedulazione del disco

Durante gli ultimi 30 anni, l'aumento della velocità dei processori e della memoria centrale ha abbondantemente superato quello dell'accesso a disco: le velocità di processori e memoria centrale aumentano di circa due ordini di grandezza, nello stesso tempo in cui la velocità del disco aumenta di un ordine di grandezza. Il risultato è che i dischi sono attualmente almeno di quattro ordini di grandezza più lenti della memoria centrale, e questo dislivello aumenterà nel prossimo futuro. Di conseguenza, le prestazioni dei sistemi di gestione dei dischi sono d'importanza centrale, e molto lavoro di ricerca è volto a produrre schemi che migliorino queste prestazioni. In questa sezione, sottolineeremo alcuni elementi chiave e descriveremo gli approcci più importanti. Siccome le prestazioni del disco sono strettamente legate agli aspetti di progettazione dei file, la discussione continuerà nel Capitolo 12.

Parametri di prestazioni del disco

I dettagli effettivi delle operazioni di I/O del disco dipendono dal sistema di elaborazione, dal sistema operativo e dalla natura del canale di I/O e dell'hardware del controller del disco; un diagramma di tempo generale è mostrato in Figura 11.6.

Quando il drive del disco sta operando, il disco ruota a velocità costante. Per leggere o scrivere, la testina deve essere posizionata all'inizio del settore desiderato della particolare traccia desiderata. Per scegliere la traccia, in un sistema a testina mobile la testina deve essere mossa, mentre, in un sistema a testina fissa, si deve selezionare elettronicamente una particolare testina; il tempo richiesto per posizionare la testina sulla traccia è noto come **tempo di ricerca** (seek time). In entrambi i casi, una volta che la traccia è stata selezionata, il controller del disco attende finché il settore interessato non si allinea con la testina; il tempo richiesto affinché il settore raggiunga la testina è noto come **ritardo di rotazione** (rotational delay), o latenza di rotazione. La somma di tempo di ricerca, nel caso ci sia, e di ritardo di rotazione è detta **tempo di accesso** (access time) e rappresenta il tempo necessario per raggiungere la posizione di lettura o scrittura. Nel momento in cui la testina è in posizione, l'operazione di lettura o scrittura può essere effettuata mentre il settore si muove al di sotto della testina; questa è la porzione di trasferimento dati dell'operazione.

In aggiunta al tempo di accesso e di trasferimento, esistono svariati ritardi normalmente associati alle operazioni di I/O del disco. Quando un processo emette una richiesta di I/O, deve

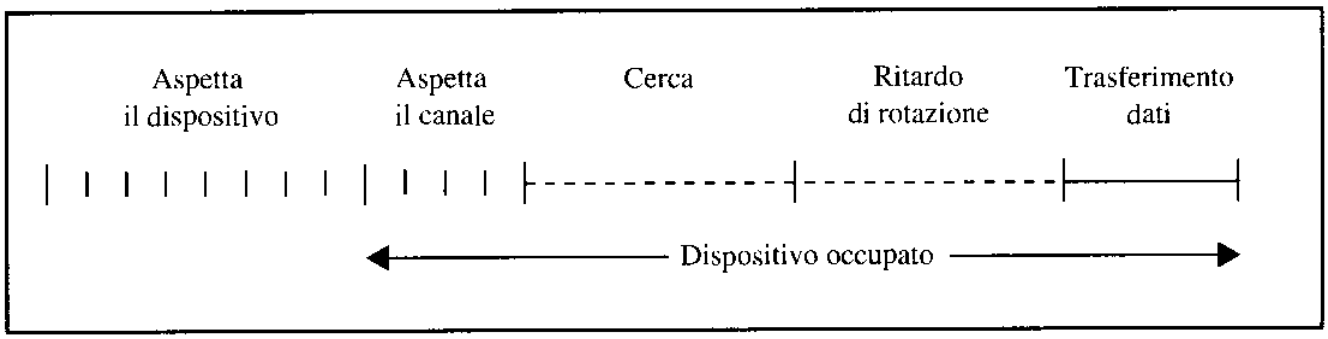


Figura 11.6 *Temporizzazione di un trasferimento I/O del disco*

prima di tutto mettersi in coda e aspettare che il dispositivo sia disponibile; quando questo accade, il dispositivo viene assegnato al processo. Se il dispositivo condivide un canale di I/O o un insieme di canali di I/O con altri drive del disco, può esserci un'attesa ulteriore affinché il canale diventi disponibile; a questo punto la ricerca e l'accesso al disco possono essere effettuati.

In alcuni sistemi mainframe, si usa una tecnica nota come RPS (*rilevamento rotazionale posizionale,* Rotational Positional Sensing), che funziona nel seguente modo: quando si invia un comando di ricerca, il canale è rilasciato per permettere la gestione di altre operazioni di I/O. Quando la ricerca è completata, il dispositivo determina quando i dati ruoteranno sotto la testina; man mano che il settore si avvicina alla testina, il dispositivo cerca di ristabilire il cammino di comunicazione con l'ospite. Se l'unità di controllo o il canale sono occupati con un altro I/O, il tentativo di riconnessione fallisce, e il dispositivo deve ruotare di una rivoluzione completa prima di poter ritentare la riconnessione; questo è detto perdita di un RPS, ed è un ulteriore elemento di ritardo che deve essere aggiunto alla linea del tempo in Figura 11.6.

Tempo di ricerca

Il tempo di ricerca è il tempo necessario per muovere il braccio del disco sulla traccia richiesta; questa quantità risulta difficile da specificare. Il tempo di ricerca si compone di più elementi: il tempo di avvio (startup time), ed il tempo richiesto per percorrere i cilindri che devono essere attraversati dal momento in cui il braccio di accesso è in moto a regime. Quest'ultimo, sfortunatamente, non è lineare nel numero delle tracce. Possiamo approssimare il tempo di ricerca con la seguente formula lineare

$$T_s = m \times n + s$$

dove

T_s = tempo di ricerca stimato

n = numero di tracce attraversate

m = costante che dipende dal drive del disco

s = tempo di avvio

Per esempio, un disco fisso economico su un personal computer potrebbe essere approssimato con $m = 0.3$ ms e $s = 20$ ms, mentre un drive più grande e costoso potrebbe avere $m = 0.1$ ms e $s = 3$ ms.

Ritardo di rotazione

A parte i floppy disk, i dischi normalmente ruotano a 3600 rpm, ossia una rivoluzione ogni 16.7 ms, pertanto, in media, il ritardo di rotazione è di circa 8.3 ms. I floppy disk tipicamente ruotano tra i 300 e i 600 rpm, quindi il ritardo medio sarà tra i 100 e i 200 ms.

Tempo di trasferimento

Il tempo di trasferimento verso o da disco, dipende dalla velocità di rotazione del disco nel seguente modo:

$$T = \frac{b}{rN}$$

dove

T = tempo di trasferimento

b = numero di byte da trasferire

N = numero di byte su una traccia

r = velocità di rotazione, in rivoluzioni al secondo

Pertanto il tempo di accesso medio totale può essere espresso come

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

dove T_s è il tempo di ricerca medio.

Un confronto sui tempi

Con i parametri appena definiti, analizziamo due diverse operazioni di I/O che illustrano il pericolo di fidarsi dei valori medi. Consideriamo un disco tipico con un tempo di ricerca medio dichiarato di 20ms, un tempo di trasferimento di 1 MB/s, settori di 512 byte e 32 settori per traccia. Supponiamo di voler leggere un file composto da 256 settori, per un totale di 128 KB; vogliamo stimare il tempo totale necessario al trasferimento.

Prima di tutto supponiamo che il file sia memorizzato nel modo più compatto possibile su disco: il file occupa tutti i settori su 8 tracce adiacenti (8 tracce × 32 settori per traccia = 256 settori); questa è detta *organizzazione sequenziale*. Quindi, il tempo necessario a leggere la prima traccia è il seguente:

Ricerca media	20	ms
Ritardo di rotazione	8.3	ms
Lettura di 32 settori	16.7	ms
	45	ms

Supponiamo che le rimanenti tracce possano essere lette essenzialmente senza ulteriore bisogno di ricerca, ossia, che l'operazione di I/O si mantenga al passo con il flusso di dati dal disco; al massimo, avremo a che fare con il ritardo di rotazione di ogni traccia successiva, pertanto ogni traccia successiva viene letta in 8.3 + 16.7 = 25ms. Per leggere l'intero file,

$$\text{Tempo totale} = 45 + 7 \times 25 = 220 \text{ ms} = 0.22\text{s}$$

Adesso calcoliamo il tempo richiesto per leggere gli stessi dati usando un accesso casuale invece di un accesso sequenziale; ossia, gli accessi ai settori sono distribuiti in modo casuale sul disco. Per ogni settore, quindi, abbiamo

Ricerca media	20	ms
Ritardo di rotazione	8.3	ms
Lettura di 1 settore	0.5	ms
	28.8	ms

$$\text{Tempo totale} = 256 \times 28.8 = 7373 \text{ ms} = 7.37 \text{ s}$$

È chiaro che l'ordine in cui i settori vengono letti dal disco ha un effetto significativo sulle prestazioni dell'I/O; in caso di accesso a file in cui molti settori vengono letti o scritti, abbiamo un certo controllo sul modo in cui settori di dati sono posizionati, e avremo qualcosa da dire a questo riguardo nel prossimo capitolo. Comunque, anche nel caso di accesso a un file, in un ambiente di multiprogrammazione, ci saranno richieste di I/O che competeranno per lo stesso disco; pertanto, è il caso di esaminare modi per migliorare le prestazioni dell'I/O del disco, rispetto a quanto ottenuto con un semplice accesso casuale al disco.

Politiche di schedulazione del disco

Nell'esempio appena descritto, la ragione delle diverse prestazioni è dovuta al tempo di ricerca: se le richieste di accesso a settori selezionano tracce a caso, le prestazioni del sistema di I/O del disco saranno molto scarse. Per migliorare le cose, dobbiamo ridurre il tempo medio speso nelle ricerche.

Consideriamo la tipica situazione di un ambiente in multiprogrammazione, in cui il sistema operativo mantiene una coda di richieste per ogni dispositivo di I/O; quindi, per un singolo disco, ci sarà un numero di richieste di I/O (letture e scritture) provenienti da vari processi in coda. Se selezionassimo gli elementi dalla coda in modo casuale, dovremmo aspettarci che le tracce siano visitate in modo casuale, producendo le prestazioni peggiori possibili: questa **schedulazione casuale** è un utile termine di paragone rispetto al quale valutare altre tecniche.

La più semplice forma di schedulazione è quella **FIFO** (*primo entrato primo uscito*, First-In-First-Out), che esegue semplicemente le richieste messe in coda in modo sequenziale; tale strategia ha il vantaggio di essere equa, in quanto ogni richiesta viene onorata, e le richieste sono onorate nell'ordine ricevuto. La Figura 11.7a illustra i movimenti del braccio del disco con il FIFO. In questo esempio, supponiamo che il disco abbia 200 tracce, e che la coda di richieste del disco contenga richieste casuali: le tracce richieste, in ordine di ricevimento, sono 55, 58, 39, 18, 90, 160, 38, 184. La Tabella 11.3a raggruppa i risultati.

Con FIFO, se vi sono solo pochi processi che richiedono l'accesso, e se molte richieste riguardano settori ravvicinati, possiamo augurarci buone prestazioni, però questa tecnica offrirà spesso prestazioni vicine a quelle dello schedulazione casuale, quando molti processi competono per il disco; di conseguenza, occorrerà considerare politiche di schedulazione più sofisticate. Alcune di esse sono elencate nella Tabella 11.4 e sono descritte di seguito.

Tabella 11.3 *Confronto tra algoritmi di schedulazione del disco*

(a) FIFO Iniziando alla traccia 100		(b) SSTF Iniziando alla traccia 100		(c) SCAN Iniziando alla traccia 100, muovendosi in direzione dei numeri di traccia crescenti		(c) CSCAN Iniziando alla traccia 100, muovendosi in direzione dei numeri di traccia crescenti	
Prossima traccia a cui si accede	**Numero di tracce attraversate**	**Prossima traccia a cui si accede**	**Numero di tracce attraversate**	**Prossima traccia a cui si accede**	**Numero di tracce attraversate**	**Prossima traccia a cui si accede**	**Numero di tracce attraversate**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Lunghezza media di ricerca	55.3	**Lunghezza media di ricerca**	27.5	**Lunghezza media di ricerca**	27.8	**Lunghezza media di ricerca**	35.8

Tabella 11.4 *Algoritmi di schedulazione del disco [WIED87]*

Nome	Descrizione	Note
	Selezione secondo il richiedente	
RSS	Schedulazione casuale	Per analisi e simulazione
FIFO	Primo a entrare primo a uscire	Il più equo di tutti
PRI	Priorità di processo	Controllo al di fuori della gestione di coda al disco
LIFO	Ultimo a entrare primo a uscire	Massimizza la località e l'utilizzazione delle risorse
	Selezione secondo l'elemento richiesto	
SSTF	Prima il tempo di servizio minore	Alta utilizzazione, code brevi
SCAN	Avanti e indietro sul disco	Migliore distribuzione dei servizi
C-SCAN	Senso unico con ritorno veloce	Minore variabilità del servizio
N-step-SCAN	SCAN di N record alla volta	Servizio garantito
FSCAN	N-step-SCAN con N=dimensione coda all'inizio del ciclo di SCAN	Sensibile al carico

Priorità

Con un sistema basato sulle priorità (PRI), il controllo della schedulazione ricade al di fuori del controllo del software di gestione del disco: con tale approccio non si vuole ottimizzare l'utilizzo del disco, ma si vogliono raggiungere altri obiettivi relativi al sistema operativo. Infatti, spesso job brevi in batch e job interattivi ricevono priorità maggiore di job più lunghi che richiedono un maggior tempo di calcolo, in modo da eliminare velocemente molti piccoli job ed ottenere un buon tempo di risposta interattiva, ma job più lunghi possono dover attendere per tempi troppo lunghi. Tale politica potrebbe portare a contromisure da parte degli utenti, i quali potrebbero dividere i loro job in piccoli pezzi per battere il sistema; inoltre, questa politica tende ad essere poco interessante per sistemi di basi di dati.

LIFO (ultimo entrato primo uscito, Last In First Out)

Sorprendentemente, la politica di eseguire sempre l'ultima richiesta arrivata ha alcuni meriti: in sistemi per l'elaborazione di transazioni, dare il dispositivo all'utente più recente minimizza i movimenti del braccio nel muoversi all'interno di un file sequenziale; trarre vantaggio dalla località migliora il throughput e diminuisce la lunghezza delle code. Finché un job può usare attivamente il file system, viene eseguito il più velocemente possibile, ma se il disco è mantenuto occupato a causa di un grande carico di lavoro, è possibile raggiungere uno stato di starvation. Nel momento in cui il job ha inserito una richiesta di I/O in coda ed ha abbandonato la testa della coda, esso può non ritornare più in testa, a meno che la coda di fronte a lui si svuoti.

La schedulazione FIFO, con priorità e LIFO si basa unicamente sugli attributi della coda o del richiedente. Se la posizione della traccia corrente è nota allo schedulatore, allora si può effettuare una schedulazione basata sull'elemento richiesto. Nel seguito esamineremo questa politica.

SSTF (prima il tempo di servizio minore, Shortest Service Time First)

La politica SSTF è quella di selezionare la richiesta di I/O con il disco che richiede il minor movimento del braccio del disco dalla sua posizione corrente, ossia si sceglie sempre il tempo di ricerca minore. Ovviamente, scegliere sempre il tempo di ricerca minore non garantisce che il tempo di ricerca medio, su un certo numero di movimenti del braccio, sia minimo, comunque questo approccio dovrebbe garantire migliori prestazioni rispetto a FIFO. Siccome il braccio può muoversi in due direzioni, un algoritmo di scelta casuale può essere utilizzato per risolvere i casi di distanze uguali.

La Figura 11.7b e la Tabella 11.3b mostrano le prestazioni di SSTF sullo stesso esempio usato per FIFO.

SCAN

Ad eccezione di FIFO, tutte le politiche descritte fino a questo momento lasciano alcune richieste insoddisfatte finché l'intera coda non viene svuotata; questo significa che possono esserci sempre nuove richieste in arrivo che verranno scelte prima di quelle già esistenti. Una semplice alternativa che può impedire la starvation è l'algoritmo SCAN.

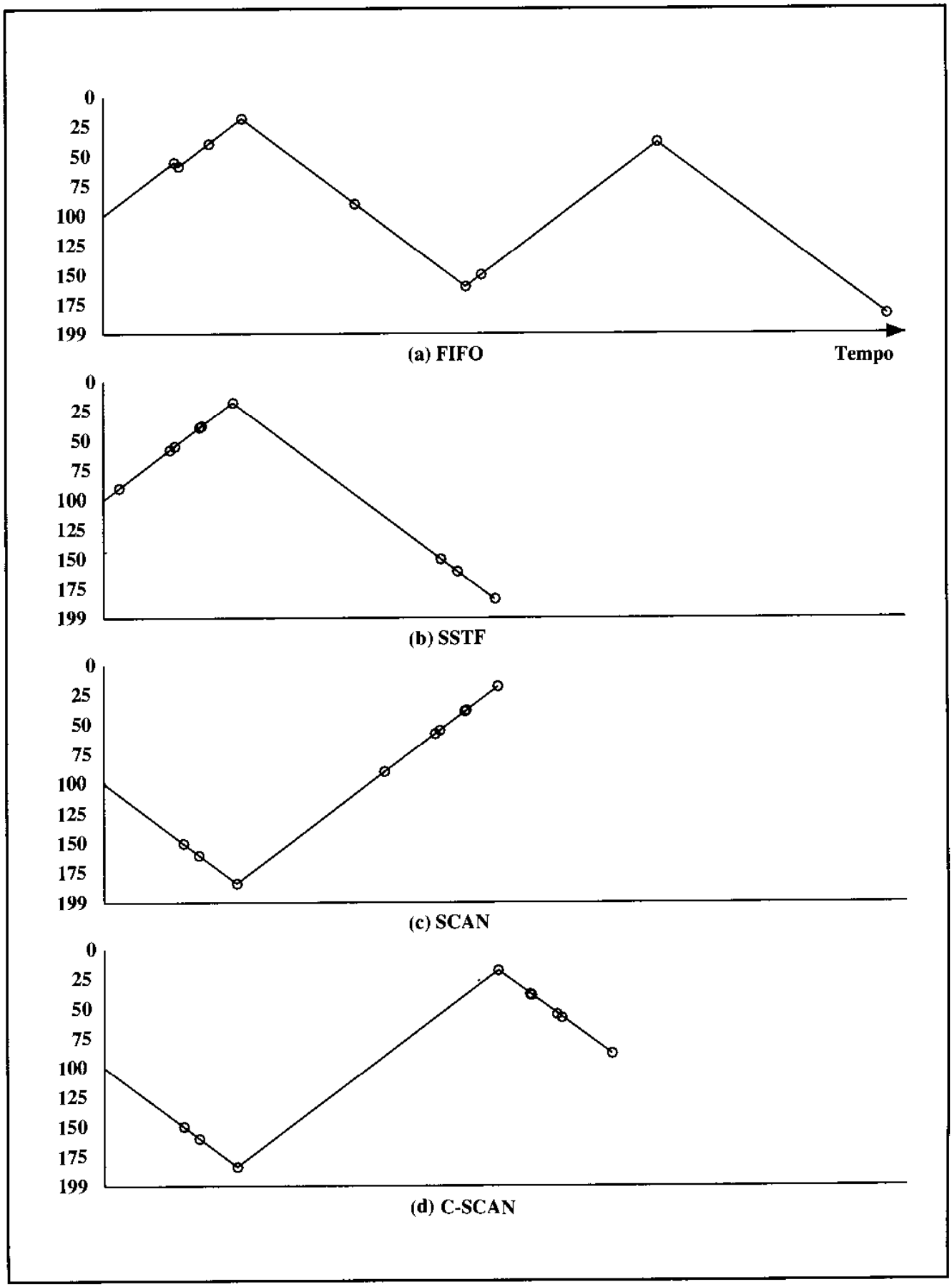


Figura 11.7 *Confronto tra algoritmi di schedulazione del disco (vedi Tabella 11.3)*

Con lo SCAN, il braccio può muoversi solo in una direzione, soddisfacendo tutte le richieste in attesa sulla sua strada, finché non raggiunge l'ultima traccia in quella direzione o finché non ci sono più richieste in quella direzione; quest'ultima strategia viene spesso chiamata LOOK. La direzione di servizio viene quindi invertita, e la coda viene scandita nella direzione opposta, anche questa volta scegliendo le richieste in ordine.

La Figura 11.7c e la Tabella 11.3c illustrano la politica SCAN. Si può notare che essa si comporta in modo quasi identico alla politica SSTF, infatti, se avessimo formulato l'ipotesi che il braccio si stava muovendo verso i numeri di traccia minori all'inizio dell'esempio, allora i cammini di schedulazione sarebbero stati identici per SSTF e SCAN; comunque, questo è un esempio statico, nel quale non vengono inserite nuove tracce nella coda. Anche se la coda stesse cambiando in modo dinamico, SCAN sarebbe simile a SSTF, a meno che lo schema di richiesta sia insolito.

Notare che la politica SCAN è sfavorevole all'area attraversata più di recente; pertanto non sfrutta la località così bene come SSTF o persino LIFO.

Non è difficile notare che la politica SCAN è favorevole ai job che richiedono tracce vicine ai cilindri più esterni o più interni, e che favorisce i job più recenti; il primo problema può essere evitato con la politica C-SCAN, mentre il secondo è affrontato dalla politica N-step-SCAN.

C-SCAN

La politica C-SCAN restringe la scansione ad un'unica direzione; pertanto quando l'ultima traccia in una direzione è stata visitata, il braccio viene riportato all'estremo opposto e la scansione ricomincia da capo, riducendo quindi il ritardo massimo possibile per le nuove richieste. Con SCAN, se il tempo atteso per una scansione dalla traccia più interna alla traccia più esterna è t, l'intervallo atteso tra un servizio e l'altro per settori che si trovano alla periferia, è $2t$; con C-SCAN, l'intervallo è nell'ordine di $t+s_{max}$, dove s_{max} è il tempo di ricerca massimo.

La Figura 11.7d e la Tabella 11.3d illustrano il comportamento di C-SCAN.

N-step-SCAN e F-SCAN

Con SSTF, SCAN e C-SCAN, è possibile che il braccio non si muova per un considerevole periodo di tempo; per esempio, se uno o più processi hanno alte frequenze di accesso ad una particolare traccia, essi possono monopolizzare l'intero dispositivo ripetendo le richieste per quella traccia. Dischi ad alta densità con molte superfici sono più soggetti a questo comportamento rispetto a dischi a minore densità e/o dischi con solo una o due superfici; per evitare questo fenomeno, la coda di richieste del disco può essere segmentata e si può svuotare completamente una coda alla volta. Due esempi di questo approccio sono N-step-SCAN (N passi di SCAN) e FSCAN.

La politica N-step-SCAN segmenta la coda di richieste del disco in sottocode di lunghezza N. Le sottocode sono elaborate una alla volta, usando SCAN; mentre una coda è elaborata, le nuove richieste devono essere aggiunte alle altre code e, se alla fine di una scansione vi sono meno di N richieste disponibili, queste saranno tutte eseguite nella successiva scansione. Per grandi valori di N, le prestazioni di N-step-SCAN si avvicinano a quelle di SCAN; per N=1, viene adottata la politica FIFO.

FSCAN è una politica che usa due sottocode e, quando la scansione comincia, tutte le richieste sono in una delle due code, mentre l'altra è vuota: la si riempirà durante la scansione, con tutte le nuove richieste che arrivano. Pertanto, tutte le nuove richieste sono servite dopo tutte le vecchie richieste.

11.6 RAID

Come è stato discusso in precedenza, le percentuali di miglioramento nelle prestazioni della memoria secondaria sono sensibilmente minori di quelle dei processori e della memoria centrale; questa differenza ha reso il sistema di memorizzazione dati del disco la principale preoccupazione per chi intende migliorare le prestazioni generali di un sistema di elaborazione.

Come in altre aree di analisi di prestazioni dei computer, i progettisti dei dischi riconoscono che, se una componente non può essere spinta oltre un certo limite, gli ulteriori guadagni si potranno ottenere solo utilizzando componenti parallele; nel caso di memorizzazione su disco, questo porta a sviluppare array di dischi che operano indipendentemente ed in parallelo. Con dischi multipli, richieste di I/O separate possono essere eseguite in parallelo, sempre che i dati richiesti si trovino su dischi separati; inoltre, una singola richiesta di I/O può essere eseguita in parallelo se il blocco di dati a cui accedere è distribuito su diversi dischi.

Usando dischi multipli, i dati possono essere organizzati in vari modi, sfruttando la ridondanza per migliorare l'affidabilità; questo renderebbe difficile lo sviluppo di schemi di database che siano utilizzabili su un certo numero di piattaforme e di sistemi operativi. Fortunatamente, le industrie hanno concordato uno schema standardizzato per lo sviluppo di database su dischi multipli, chiamato RAID (*array ridondante di dischi indipendenti*, Redundant Array of Independent Disks). Lo schema RAID si compone di sei livelli,[1] dallo zero al cinque; questi livelli non sono posti in relazione gerarchica, ma identificano diverse architetture di progettazione, che condividono tre caratteristiche comuni:

1. RAID è un insieme di dischi fisici visti dal sistema operativo come un unico drive logico.
2. I dati sono distribuiti attraverso i drive fisici di un array.
3. La capacità di ridondanza del disco è usata per memorizzare informazioni di parità, che garantiscono che i dati vengano ricuperati in caso di guasto del disco.

I dettagli della seconda e della terza caratteristica differiscono a seconda dei diversi livelli RAID; RAID 0 non supporta la terza caratteristica.

[1] Livelli aggiuntivi sono stati definiti da alcuni ricercatori ed alcune compagnie, ma i sei livelli descritti in questa sezione sono quelli su cui si è universalmente d'accordo.

Il termine *RAID* fu originariamente coniato in un articolo da un gruppo di ricercatori dell'Università della California a Berkeley[2]. L'articolo delineava svariate possibili configurazioni di RAID con le possibili applicazioni e introduceva le definizioni dei livelli di RAID che sono tuttora usate. RAID si è proposto per eliminare il divario crescente tra le velocità dei processori e quelle dei drive elettromeccanici dei dischi, relativamente lenti; la strategia è quella di rimpiazzare dischi di grandi capacità con molti dischi di minore capacità, distribuendo i dati in modo che sia possibile l'accesso simultaneo ai dati, e quindi migliorino le prestazioni dell'I/O, permettendo agevolmente di aumentare le capacità.

Un contributo unico della proposta RAID è quello di gestire efficacemente la necessita di ridondanza; infatti, permettere testine multiple e attuatori che operino simultaneamente consente velocità di trasferimento più alte, ma aumenta anche la probabilità di guasto. Per compensare questa minore affidabilità, RAID conserva informazioni di parità, che permettano il ripristino di dati persi a causa del guasto di un disco.

A questo punto esamineremo ognuno dei livelli RAID; la Tabella 11.5 riassume i sei livelli. Di questi, i livelli 2 e 4 non vengono offerti commercialmente e difficilmente verranno accettati dall'industria; comunque, una descrizione di questi due livelli aiuta a chiarire le scelte progettuali di alcuni degli altri livelli.

La Figura 11.8 illustra i sei schemi RAID che supportano una capacità di dati che richiede quattro dischi senza ridondanza; la figura evidenzia la struttura dei dati utente e dei dati ridondanti, e indica i requisiti di memorizzazione relativi ai vari livelli. Faremo riferimento a questa figura per tutta la discussione seguente.

Livello 0 di RAID

Il livello 0 di RAID non è un vero componente della famiglia RAID, in quanto non comprende la ridondanza per il miglioramento delle prestazioni. Comunque, vi sono alcune applicazioni, ad esempio per i supercomputer, in cui prestazioni e capacità sono le principali preoccupazioni, e il basso costo è più importante di un aumento di affidabilità.

In RAID 0, i dati utente e i dati di sistema sono distribuiti su tutti i dischi dell'array. Questo ha un vantaggio notevole rispetto all'uso di un unico disco di maggiori dimensioni: se due diverse richieste di I/O sono in attesa di due diversi blocchi di dati, c'è una buona probabilità che i due blocchi richiesti si trovino su dischi diversi. Pertanto le due richieste possono essere emesse in parallelo, riducendo il tempo in coda per l'I/O.

Ma RAID 0, così come tutti i livelli di RAID, va al di là della distribuzione di dati in un array di dischi: i dati sono sistemati *a strisce* sui dischi disponibili, ossia si effettua quello che è detto

---

[2] In quell'articolo, l'acronimo RAID stava per Redundant Array of Inexpensive Disks (array ridondante di dischi economici). Il termine *inexpensive* venne usato per contrapporre array di dischi, piccoli e relativamente economici di RAID all'alternativa di un unico, costoso disco grande (Single Large Expensive Disk, SLED). Lo SLED è essenzialmente una cosa del passato, la tecnologia adottata sia dalle configurazioni RAID sia da quelle non-RAID è ormai la stessa; di conseguenza le industrie hanno preferito il termine *indipendenti*, per sottolineare il fatto che il vettore RAID consente prestazioni significative e guadagni nell'affidabilità.

Tabella 11.5 *I livelli di RAID*

Categoria	Livello	Descrizione	Frequenza di richieste I/O (read/write)	Tasso di trasferimento dati (read/write)	Applicazioni tipiche
Striping	0	Non ridondante	Fette larghe:eccellente	Fette piccole: eccellente	Applicazioni che richiedono alte prestazioni su dati non critici
Duplicazione	1	Duplicato	Buona/discreta	Discreta/discreta	Drive di sistema; file critici
Accesso parallelo	2	Ridondante tramite codifica di Hamming	Scarsa	Eccellente	
	3	Parità bit per bit	Scarsa	Eccellente	Richieste di I/O grandi: applicazioni su immagini, CAD
Accesso indipendente	4	Parità blocco per blocco	Eccellente/discreta	Discreta/scarsa	
	5	Parità distribuita blocco per blocco	Eccellente/discreto	Discreto/scarsa	Alte frequenze di richieste, con molte letture, ricerche sui dati

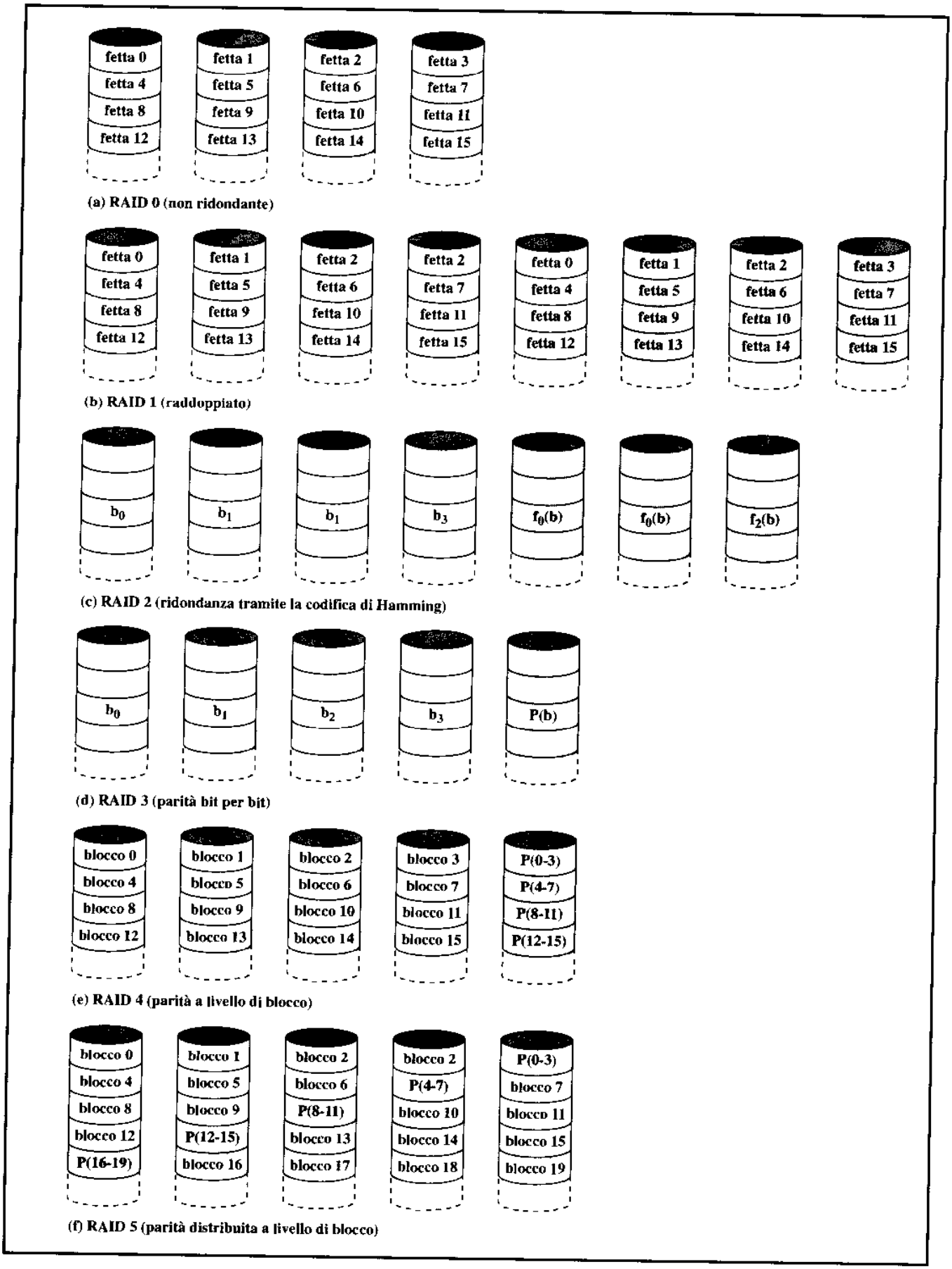


Figura 11.8 *I livelli di RAID*

lo striping di dati. Questo concetto può essere capito meglio considerando la Figura 11.9; tutti i dati utente e di sistema sono visti come se fossero memorizzati in un disco logico. Questo disco è diviso in fette (strip), che possono essere blocchi fisici, settori o altre unità; le fette sono mappate round robin su elementi consecutivi dell'array di dischi. Un insieme di fette logicamente consecutive che mappi esattamente una fetta in ogni elemento dell'array è detto *striscia* (stripe).

In un'array di n dischi, le prime n fette logiche sono memorizzate fisicamente sulla prima fetta di ciascun disco; le seconde n fette sono distribuite sulle seconde n fette di ogni disco, e così via. Il vantaggio di questo schema è che se una richiesta di I/O si riferisce a più fette logicamente contigue, allora si possono elaborare in parallelo fino a n fette, riducendo moltissimo il tempo di trasferimento.

La Figura 11.9 mostra l'uso di un software per la gestione di array, che mappa lo spazio disco logico e fisico; tale software può essere eseguito sul sottosistema di gestione del disco, o sul sistema ospite.

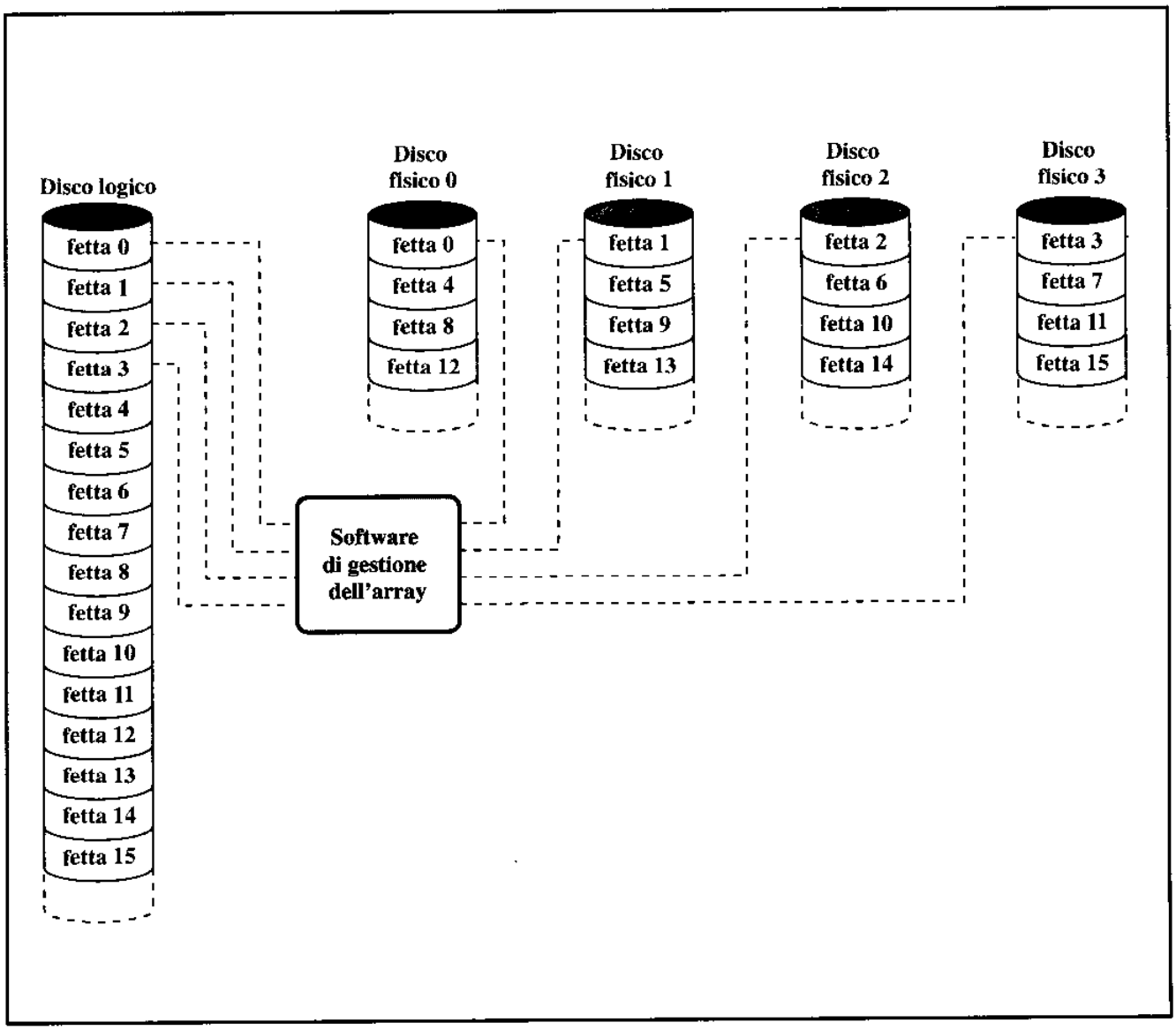


Figura 11.9 *Mapping di dati su un array RAID di livello 0 [MASS94]*

RAID 0 per un'alta capacità di trasferimento dati

Le prestazioni di ognuno dei livelli di RAID dipendono in modo critico dalla struttura delle richieste del sistema ospite e dalla struttura dei dati; questo problema può essere descritto con maggior chiarezza nel caso di RAID 0, dove l'impatto della ridondanza non interferisce con l'analisi. Prima di tutto, consideriamo l'uso di RAID 0 per ottenere un altro tasso di trasferimento; perché le applicazioni ne possano godere, vi sono due requisiti. Prima di tutto, ci deve essere un'alta capacità di trasferimento lungo tutto il percorso tra la memoria del sistema ospite e i singoli drive del disco, compresi i controller di bus interni, i bus di I/O del sistema ospite, gli adattatori di I/O e il bus di memoria del sistema ospite.

Il secondo requisito è che l'applicazione deve effettuare richieste di I/O che sfruttino in modo efficiente l'array dei dischi; questo requisito è soddisfatto se tipicamente si richiedono grandi quantità di dati logicamente contigui, rispetto alla dimensione della fetta. In questo caso, una singola richiesta di I/O riguarda il trasferimento in parallelo di dati da vari dischi, aumentando l'effettivo tasso di trasferimento rispetto ad un trasferimento da singolo disco.

RAID 0 per un alto tasso di richieste di I/O

In un ambiente orientato alle transazioni, tipicamente l'utente si preoccupa più del tempo di risposta che del tempo di trasferimento; in una singola richiesta di I/O di una piccola quantità di dati, il tempo di I/O è dominato dal movimento delle testine (tempo di ricerca) e dal movimento del disco (latenza di rotazione).

In un ambiente a transazioni, possono esserci centinaia di richieste di I/O al secondo; un array di dischi può fornire un alto tasso di esecuzione di I/O, bilanciando il carico di I/O attraverso molti dischi, e un efficace bilanciamento del carico si ottiene solo se ci sono più richieste di I/O in attesa. Questo implica che ci siano molte applicazioni indipendenti, oppure un'unica applicazione basata su transazioni, che effettua molte richieste di I/O asincrone. Le prestazioni saranno ulteriormente influenzate dalla dimensione delle fette; se essa è relativamente grande, una singola richiesta di I/O richiederà un solo accesso a disco, ed allora molte richieste di I/O in attesa possono essere servite in parallelo, riducendo il tempo di accodamento per ogni richiesta.

Livello 1 di RAID

RAID 1 differisce da RAID ai livelli da 2 a 5 per il modo in cui la ridondanza viene raggiunta: negli schemi RAID superiori, si usa il calcolo della parità per realizzare la ridondanza, mentre in RAID 1 la ridondanza si ottiene semplicemente duplicando tutti i dati. Come mostra la Figura 11.8b, viene usato lo striping di dati, così come in RAID 0; in questo caso però, ogni fetta logica viene mappata in due dischi fisici separati, di modo che ogni disco dell'array ha un disco gemello che contiene gli stessi dati.

Vi sono molti aspetti positivi nell'organizzazione di RAID 1:

1. Una richiesta di lettura può essere servita da uno qualunque dei due dischi che contengono i dati richiesti, quello che richiede minor tempo di ricerca e latenza di rotazione.

2. Una richiesta di scrittura richiede che entrambe le fette corrispondenti siano aggiornate, ma questo può essere fatto in parallelo; pertanto le prestazioni dell'operazione di scrittura sono legate a quella della più lenta delle due scritture (cioè quella che richiede un tempo di ricerca e una latenza di rotazione più lunghi); in ogni caso, non c'è alcuna "penalità di scrittura" in RAID 1. I livelli da 2 a 5 di RAID utilizzano bit di parità, quindi, quando una singola fetta viene aggiornata, il software di gestione dell'array deve anche calcolare e aggiornare i bit di parità, oltre ad aggiornare la fetta in questione.
3. Il recupero da un guasto è semplice: quando un drive fallisce, i dati possono essere recuperati dal secondo.

Il principale svantaggio di RAID 1 è il costo; richiede il doppio dello spazio disco del disco logico che supporta. Per questo motivo, una configurazione di RAID 1 è praticamente limitata ai drive che conservano il software e i dati di sistema, ed altri file altamente critici; in questi casi, RAID 1 fornisce un backup di tutti i dati in tempo reale, in modo che, in caso di guasto del disco, tutti i dati critici siano ancora immediatamente disponibili.

In ambienti orientati alle transazioni, RAID 1 può raggiungere alti tassi di richieste di I/O se il grosso delle richieste sono letture; in questa situazione, le prestazioni di RAID 1 si avvicinano al doppio delle prestazioni di RAID 0; invece, se una porzione significativa delle richieste di I/O sono richieste di scrittura, possono non esserci significativi guadagni di prestazioni rispetto a RAID 0. RAID 1 può fornire miglioramenti di prestazioni rispetto a RAID 0 anche nel caso di applicazioni caratterizzate da un intensivo trasferimento di dati con un'alta percentuale di letture; i miglioramenti si ottengono se l'applicazione è in grado di separare ogni richiesta, in modo che entrambi i dischi possono partecipare.

Livello 2 di RAID

I livelli 2 e 3 di RAID fanno uso di una tecnica di accesso parallelo. In un array ad accesso parallelo, tutti i dischi partecipano a ogni richiesta di I/O; tipicamente, i perni dei drive sono sincronizzati in modo che tutte le testine dei dischi si trovino nella stessa posizione in ogni istante di tempo.

Come in altri schemi RAID, si usa lo striping dei dati e, nel caso di RAID 2 e 3, le fette sono molto piccole, spesso di un byte o una parola. In RAID 2, il codice di correzione di errori viene calcolato sui bit corrispondenti su ogni disco, e i bit del codice sono memorizzati nelle corrispondenti posizioni in dischi di parità multipli. Tipicamente, viene usato un codice Hamming, il quale è in grado di correggere errori di bit singoli e rilevare errori di bit doppi.

Anche se RAID 2 richiede meno dischi di RAID 1, è ugualmente piuttosto costoso, in quanto il numero di dischi ridondanti richiesto è proporzionale al logaritmo del numero di dischi di dati; per una singola lettura, si accede a tutti i dischi simultaneamente. I dati richiesti e il codice di correzione errori associato sono inviati al controller dell'array; se vi è un singolo bit errato, il controller può riconoscerlo e correggerlo istantaneamente, senza rallentare gli accessi in lettura. Per una singola scrittura, si accede a tutti i dischi di dati e ai dischi di parità.

RAID 2 sarebbe una scelta significativa solo in un ambiente in cui si verificano molti errori di disco, ma data l'alta affidabilità dei singoli dischi e dei drive di disco, RAID 2 è eccessivo e non viene implementato.

Livello 3 di RAID

RAID 3 è organizzato in modo simile a RAID 2, la differenza è che RAID 3 richiede un solo disco ridondante, indipendentemente dalla dimensione dell'array di dischi. RAID 3 utilizza l'accesso parallelo e i dati sono distribuiti in fette piccole. Invece di un codice di correzione di errore, si calcola un semplice bit di parità per l'insieme di bit che si trovano in posizioni corrispondenti lungo i dischi di dati.

Ridondanza

Nel caso di guasto di un drive, si accede al drive di parità e i dati vengono ricostruiti a partire dai dispositivi rimanenti; quando il drive che ha fallito è stato rimpiazzato, i dati mancanti possono essere reinseriti nel drive nuovo e l'operazione può riprendere.

La ricostruzione di dati è piuttosto semplice: consideriamo un array di cinque drive, dove i drive da X0 a X3 contengono dati, mentre X4 è un disco di parità. La parità per il bit i-esimo viene calcolata nel seguente modo:

$$X4\ (i) = X3\ (i) \oplus X2\ (i) \oplus X1\ (i) \oplus X0\ (i)$$

Supponiamo che il drive X1 fallisca: se aggiungiamo X4 $(i) \oplus$ X1 (i) a entrambi i lati della precedente equazione, otteniamo

$$X1\ (i) = X4\ (i) \oplus X3\ (i) \oplus X2\ (i) \oplus X0\ (i)$$

Pertanto, il contenuto di ogni fetta di dati in ognuno dei dischi dell'array può essere rigenerato a partire dai contenuti delle fette corrispondenti nei dischi rimanenti dell'array. Questo principio vale per i livelli 3, 4 e 5 di RAID.

Nel caso di guasto del disco, tutti i dati sono ancora disponibili in quella che viene chiamata modalità ridotta; in questa modalità, nel caso di letture, i dati mancanti sono rigenerati al volo calcolando l'or esclusivo; quando si scrivono dati in un array RAID 3 ridotto, si deve mantenere la consistenza di parità, per permettere una successiva rigenerazione; il ritorno a una piena operatività, richiede che il disco che ha fallito venga rimpiazzato, e che l'intero contenuto del disco fallito venga rigenerato e copiato sul disco nuovo.

Prestazioni

Siccome i dati sono divisi in fette molto piccole, RAID 3 può raggiungere altissimi tassi di trasferimento dati; ogni richiesta di I/O conterrà il trasferimento in parallelo di dati da tutti i dischi dell'array. Il miglioramento è notevole, specialmente nel caso di grandi trasferimenti, ma d'altro canto, solo una richiesta di I/O alla volta può essere eseguita, pertanto, in un ambiente orientato alle transazioni, le prestazioni ne soffrono.

Livello 4 di RAID

I livelli 4 e 5 di RAID utilizzano una tecnica di accesso indipendente; in un array ad accesso indipendente, ogni disco opera in modo indipendente, cosicché richieste di I/O separate possono essere espletate in parallelo. Grazie a questa caratteristica, array ad accesso parallelo sono molto adatti per applicazioni che richiedono alti tassi di richieste di I/O, e sono relativamente meno adatti per applicazioni che richiedono alti tassi di trasferimento dati.

Come negli altri schemi RAID, viene utilizzato lo striping di dati e nel caso di RAID 4 e 5, le fette sono relativamente grandi. Con RAID 4, viene calcolata una fetta di parità bit a bit tra tutte le fette corrispondenti in ogni disco di dati, e i bit di parità vengono inseriti nella posizione opportuna della fetta di parità contenuta nel disco di parità.

RAID 4 è poco vantaggioso in scrittura, quando viene eseguita una richiesta di piccole dimensioni: ogni volta che avviene una scrittura, il software di gestione dell'array deve aggiornare non solo i dati utente, ma anche i corrispondenti bit di parità. Consideriamo un array di cinque dischi in cui i dischi da X0 a X3 contengono i dati, mentre X4 è il disco di parità. Supponiamo che una scrittura riguardi una fetta presente sul disco X1. Inizialmente, per ogni bit i, abbiamo la seguente relazione:

$$X4\ (i) = X3\ (i) \oplus X2\ (i) \oplus X1\ (i) \oplus X0\ (i)$$

Dopo l'aggiornamento, indicando i bit potenzialmente alterati con un apice:

$$\begin{aligned} X4'\ (i) &= X3\ (i) \oplus X2\ (i) \oplus X1'(i) \oplus X0\ (i) \\ &= X3\ (i) \oplus X2\ (i) \oplus X1'(i) \oplus X0\ (i) \oplus X1\ (i) \oplus X1\ (i) \\ &= X4\ (i) \oplus X1\ (i) \oplus X1'(i) \end{aligned}$$

Per calcolare la nuova parità, il software di gestione dell'array deve leggere la vecchia fetta utente e la vecchia fetta di parità; poi le aggiorna ambedue con i nuovi dati e la parità appena calcolata: ogni scrittura di una fetta si compone quindi di due letture e due scritture.

Nel caso di una scrittura di I/O di maggiori dimensioni che riguardi fette appartenenti a tutti i dischi, la parità viene facilmente calcolata utilizzando solo i nuovi bit di dati; pertanto, il drive di parità può essere aggiornato in parallelo con i drive di dati e senza il bisogno di letture e scritture extra.

In ogni caso, ogni operazione di scrittura deve coinvolgere il disco di parità, che diventa quindi un collo di bottiglia.

Livello 5 di RAID

RAID 5 è organizzato in modo simile a RAID 4. La differenza è che RAID 5 distribuisce le fette di parità su tutti i dischi. Una tipica allocazione consiste nell'applicare uno schema round-robin, come illustrato in Figura 11.8f. Per un array di n dischi, la fetta di parità si trova su un disco differente per ognuna delle prime n strisce, poi la struttura si ripete.

La distribuzione delle fette di parità su tutti i drive evita il potenziale collo di bottiglia dell'I/O rilevato in RAID 4.

11.7 La cache del disco

Nella Sezione 1.6 e nell'Appendice 1A abbiamo introdotto i principi della *memoria cache*. Il termine memoria cache è solitamente usato in riferimento a una memoria, che è più piccola e più veloce della memoria centrale e che si trova tra la memoria centrale e il processore. Questa cache di memoria riduce il tempo medio di accesso in memoria, sfruttando il principio di località.

Lo stesso principio può essere applicato alla memoria secondaria sul disco; in dettaglio, una cache del disco è un buffer nella memoria centrale, che contiene una copia di alcuni settori del disco. Quando viene fatta una richiesta di I/O per un particolare settore, si controlla se questo settore si trova nella cache del disco; se sì, la richiesta è soddisfatta attraverso la cache, altrimenti, il settore richiesto viene letto dal disco e copiato nella cache. Grazie al fenomeno di località dei riferimenti, quando un blocco di dati viene portato nella cache per soddisfare una singola richiesta di I/O, molto probabilmente vi saranno riferimenti futuri allo stesso blocco.

Considerazioni sulla progettazione

Vi sono molti interessanti aspetti di progettazione della cache del disco. Prima di tutto, quando una richiesta di I/O viene soddisfatta dalla cache del disco, i dati nella cache del disco debbono essere consegnati al processo richiedente; questo può essere fatto trasferendo il blocco di dati all'interno della memoria centrale, dalla cache del disco all'area di memoria assegnata al processo utente, oppure, semplicemente, grazie alla memoria condivisa, passando un puntatore alla casella appropriata della cache del disco. Quest'ultimo approccio risparmia il tempo di un trasferimento da memoria a memoria, e permette l'accesso condiviso anche ad altri processi, secondo il modello lettore/scrittore descritto nel Capitolo 5.

Un secondo aspetto di progettazione riguarda la strategia di sostituzione; quando un nuovo settore viene portato nella cache del disco, uno dei blocchi presenti deve essere sostituito; questo è un problema identico a quello presentato nel Capitolo 8, dove si cercava un algoritmo di sostituzione di pagina. Sono stati provati molti algoritmi, quello più comunemente usato è **LRU** (*usato meno di recente*, Least Recently Used): sostituire il blocco che è rimasto più a lungo nella cache senza venire usato. Logicamente, la cache è costituita da uno stack di blocchi, con il blocco a cui ci si è riferiti più di recente in cima allo stack; quando ci si riferisce ad un blocco già presente nella cache, esso viene spostato dalla sua posizione corrente nella cache alla cima dello stack e, quando un blocco viene caricato dalla memoria secondaria, il blocco presente in fondo alla pila viene rimosso e il nuovo blocco in arrivo viene spinto in cima allo stack. Naturalmente, non è necessario muovere veramente questi blocchi in giro per la memoria; basta associare alla cache uno stack di puntatori.

Un altra possibilità è il metodo **LFU** (*meno frequentemente usato*, Least Frequently Used): sostituire il blocco nello stack che ha ottenuto il minor numero di riferimenti. Esso si potrebbe implementare associando un contatore ad ogni blocco; quando un blocco viene inserito nello stack, il suo contatore viene posto a 1, e per ogni riferimento a tale blocco il contatore viene incrementato di 1. Quando una sostituzione è necessaria, si sceglie il blocco con il contatore minore: intuitivamente, LFU sembrerebbe più opportuno di LRU, perché si usano maggiori informazioni su ogni blocco nel processo di selezione.

Un algoritmo LFU semplice ha il seguente problema: può succedere che alcuni blocchi vengano richiesti relativamente di rado, ma che quando questo accade si verifichino molti accessi ad essi in un breve intervallo di tempo, a causa del principio di località, e questi accessi multipli alzano il valore dei contatori. Dopo che questo intervallo è stato superato, il contatore può risultare fuorviante, quindi l'effetto di località può portare l'algoritmo LFU a fare scelte di sostituzioni non ottimali.

Per risolvere questa difficoltà con LFU, [ROBI90] propose una tecnica nota come sostituzione basata sulla frequenza. Per chiarezza, consideriamo prima di tutto una versione semplificata, illustrata in Figura 11.10a: i blocchi sono logicamente organizzati in uno stack, come nel caso dell'algoritmo LRU, e una certa porzione della parte superiore dello stack è messa da parte come *sezione nuova*. Quando si ha un riferimento ad un blocco già presente nella cache, questo viene portato in cima allo stack: se il blocco si trovava già nella sezione nuova, il suo contatore non viene incrementato, altrimenti è incrementato di 1. Data una sezione nuova sufficientemente grande, questo metodo non altera il contatore dei blocchi richiesti più volte in brevi intervalli di tempo. Se invece il blocco richiesto non si trova nella cache del disco, viene scelto per la sostituzione il blocco con il più piccolo valore di contatore, che non si trovi all'interno della sezione nuova; in caso di parità, si sceglie il blocco che è stato usato meno di recente.

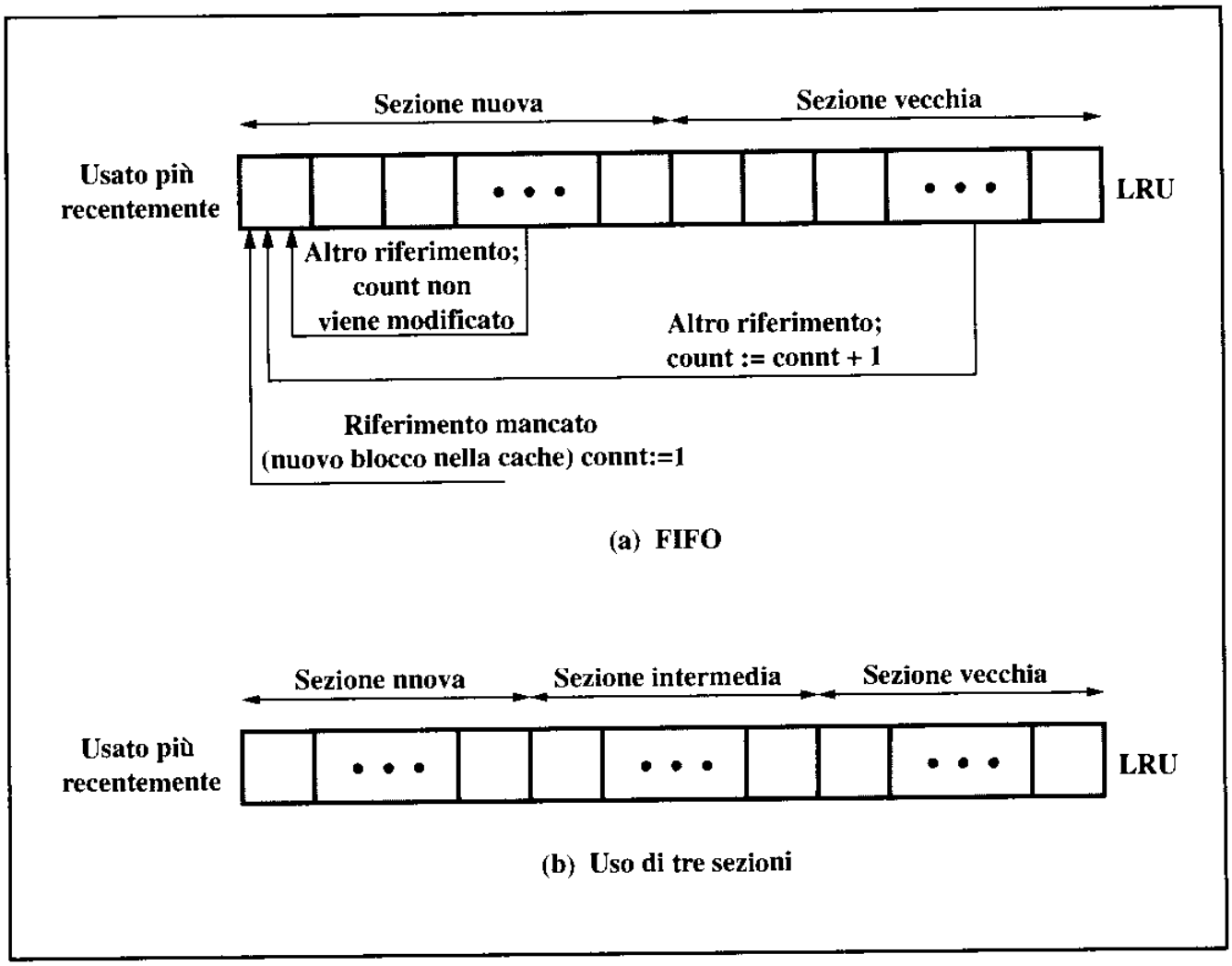


Figura 11.10 *Sostituzione basata sulla frequenza*

Gli autori affermano che con questa strategia si ottiene un miglioramento molto piccolo rispetto a LRU. Il problema è il seguente:

1. Per un blocco richiesto non presente nella cache, un nuovo blocco viene messo nella sezione nuova, con un contatore a 1.
2. Il contatore rimane a 1 finché questo blocco rimane nella sezione nuova.
3. Prima o poi il blocco sarà messo fuori dalla sezione nuova, e il suo contatore varrà sempre 1.
4. Se questo blocco non viene chiesto entro breve tempo, sarà probabilmente sostituito, in quanto il suo contatore ha il minor valore possibile. In altre parole, non sembra esserci sufficiente tempo per permettere ai blocchi che escono dalla sezione nuova di incrementare il loro contatore, anche se si accede frequentemente.

Per ovviare a questo problema si può aggiungere un ulteriore raffinamento. Lo stack viene diviso in tre sezioni: nuova, media e vecchia (Figura 11.10b); come spiegato in precedenza, i contatori non vengono incrementati per i blocchi presenti nella sezione nuova, ma solo i blocchi presenti nella sezione vecchia possono essere scelti per la sostituzione. Supponendo di avere una sezione media sufficientemente larga, questo dà la possibilità a blocchi a cui si accede frequentemente, di aumentare il loro contatore, prima di diventare potenziali elementi da sostituire. Simulazioni prodotte dagli autori, indicano che questo raffinamento è significativamente migliore di LRU o LFU.

Indipendentemente dalla possibile strategia di sostituzione, la sostituzione può avvenire a richiesta o pianificata in anticipo; nel primo caso, un settore viene sostituito solo quando lo slot è necessario, mentre nel secondo caso, vengono rilasciati un certo numero di slot per volta. Quest'ultimo approccio è legato alla necessità di riscrivere settori; se un settore è stato portato nella cache solo per essere letto, allora quando viene sostituito non deve essere riscritto su disco. Invece, se il settore è stato aggiornato, allora è necessario riscriverlo su disco prima di sostituirlo; in tal caso, è sensato raggruppare e ordinare le scritture allo scopo di minimizzare il tempo di ricerca.

Considerazioni sulle prestazioni

Possiamo applicare qui le stesse considerazioni sulle prestazioni discusse nell'Appendice 1A. La questione delle prestazioni della cache si riduce a capire se una percentuale di accessi diretti alla cache (hit ratio) può essere raggiunta; questo dipende dal comportamento della località nell'accesso al disco, dall'algoritmo di sostituzione scelto e da altri fattori di progettazione. Principalmente, la percentuale di accessi mancati alla cache, è funzione della dimensione della cache del disco stessa. La Figura 11.11 riassume i risultati ottenuti da diversi studi usando LRU, una per un sistema UNIX su un VAX [OUST85] e uno per i sistemi operativi per mainframe IBM [SMIT85]. La Figura 11.12 mostra i risultati ottenuti con simulazioni del metodo di sostituzione basato su frequenze: un confronto tra le due figure, sottolinea uno dei rischi nell'utilizzare questi metodi di analisi delle prestazioni: da esse, sembrerebbe che LRU surclassi l'algoritmo basato su frequenze, ma confrontando analoghi schemi di riferimento, usando la stessa struttura di cache, il metodo basato su frequenze è superiore.

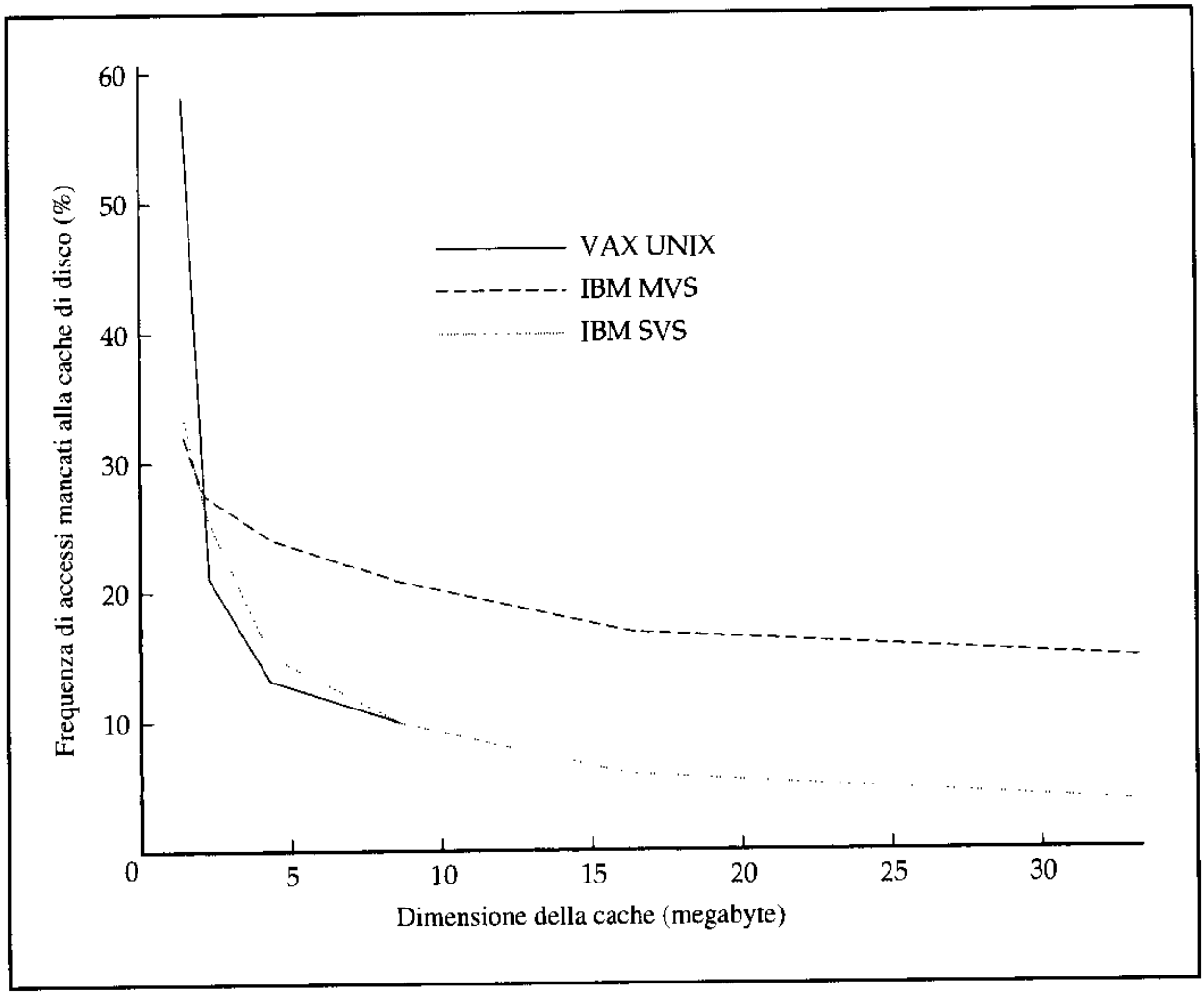


Figura 11.11 *Alcuni risultati di prestazioni di cache del disco, ottenuti usando LRU.*

Pertanto, la sequenza esatta degli schemi di riferimento scelti, più altri aspetti di progettazione quali la dimensione dei blocchi, hanno una profonda influenza sulle prestazioni raggiunta.

11.8 I/O di UNIX SVR4

In UNIX, ogni specifico dispositivo di I/O è associato ad un *file speciale*, gestito dal file system, che viene letto e scritto nello stesso modo in cui si leggono e scrivono i file di dati degli utenti, fornendo un'interfaccia pulita e uniforme a utenti e processi. Per leggere da o scrivere su un dispositivo, le richieste di lettura e scrittura vengono inviate al file speciale associato a tale dispositivo.

La Figura 11.13 illustra la struttura logica dell'I/O in UNIX: il sottosistema di file (file subsystem) gestisce i file sui dispositivi di memoria secondaria; inoltre, funge da interfaccia dei processi verso i dispositivi, in quanto questi sono trattati come file.

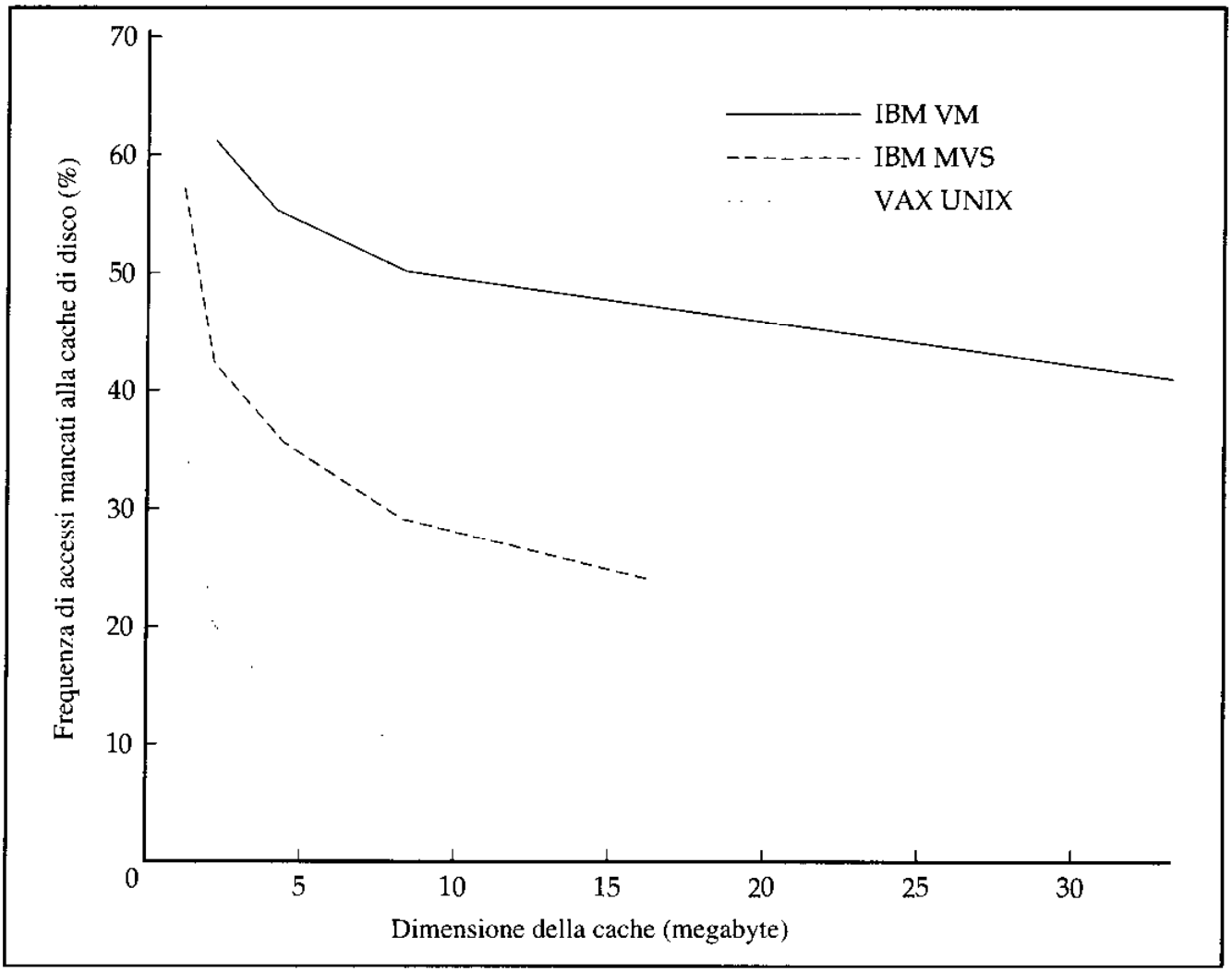


Figura 11.12 *Prestazioni di cache del disco ottenute usando l'algoritmo di sostituzione basata sulla frequenza [ROBI90]*

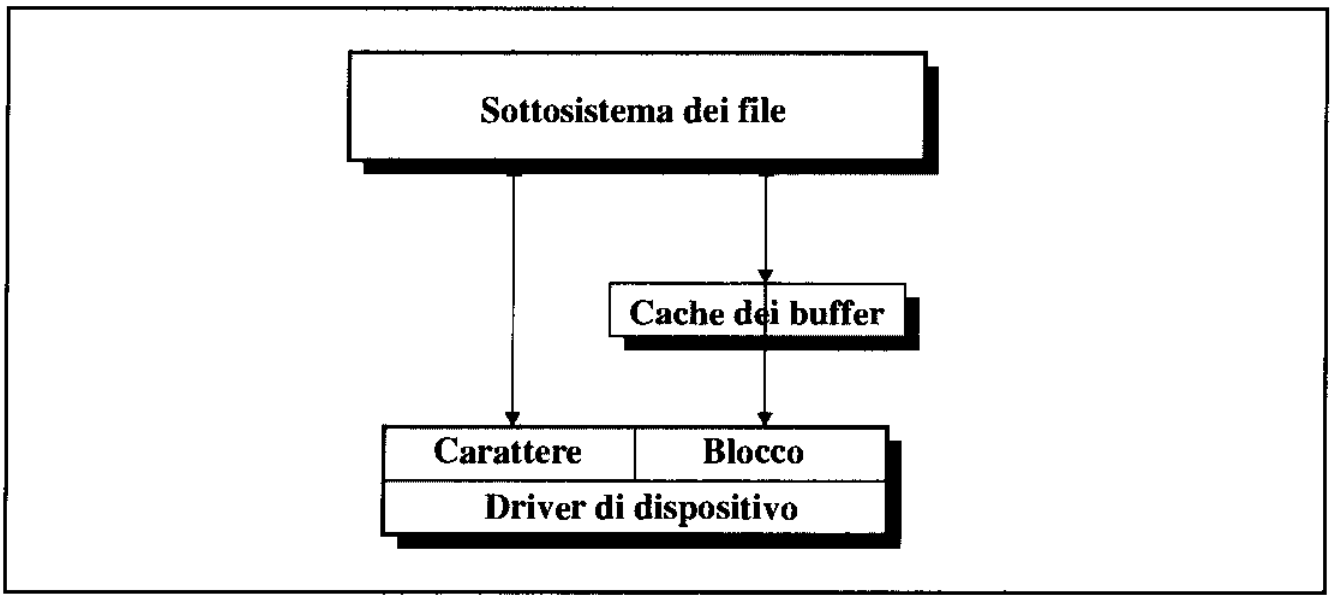


Figura 11.13 *Struttura dell'I/O di UNIX*

Esistono due tipi di I/O in UNIX: con e senza buffer; il primo passa attraverso sistemi di buffer, mentre tipicamente il secondo utilizza il DMA, e il trasferimento avviene direttamente tra il modulo di I/O e l'area di I/O del processo. Per l'I/O con buffer, due tipi di buffer vengono utilizzati: cache di buffer di sistema e code di caratteri.

Cache di buffer

In UNIX la cache di buffer è essenzialmente la cache del disco; le operazioni di I/O con il disco sono effettuate attraverso la cache di buffer. Il trasferimento di dati dalla cache di buffer allo spazio del processo utente avviene sempre utilizzando il DMA e, siccome sia la cache di buffer sia l'area di I/O del processo utente si trovano in memoria centrale, il DMA viene usato per effettuare una copia da memoria a memoria. Non vengono usati cicli del processore, ma si consumano cicli di bus.

Per gestire la cache di buffer, si mantengono tre liste:

- **Lista dei buffer liberi**: lista di tutti gli slot della cache (uno slot è un buffer in UNIX; ogni slot contiene un settore del disco) che sono disponibili per l'allocazione.
- **Lista dei dispositivi**: lista di tutti i buffer attualmente associati ad ogni disco.
- **Code dei driver I/O**: lista di buffer che stanno effettuando o stanno aspettando un I/O su un particolare dispositivo.

Tutti i buffer dovrebbero essere sulla lista dei buffer liberi, oppure sulla coda dei driver di I/O. Un buffer, una volta associato ad un dispositivo, rimane associato a quel dispositivo anche quando viene riportato nella lista dei buffer liberi, finché non viene effettivamente riutilizzato, e quindi viene associato ad un altro dispositivo. Queste liste vengono mantenute come puntatori associati ad ogni buffer, piuttosto che come liste fisicamente separate.

Quando viene effettuato un riferimento ad un numero di blocco fisico su un particolare dispositivo, il sistema operativo prima di tutto controlla che questo blocco non si trovi nella cache di buffer; per minimizzare il tempo di ricerca, la lista di dispositivi è organizzata come tabella hash, utilizzando una tecnica simile al debordamento a catena discusso nell'Appendice 8A (Figura 8.24b). La Figura 11.14 descrive l'organizzazione generale della cache di buffer: una tabella hash, di lunghezza fissata, contiene i puntatori alla cache di buffer, quindi ogni riferimento a un elemento (numero del dispositivo, numero del blocco) viene mappato in una particolare entry della tabella hash. Il puntatore contenuto in questa entry punta al primo buffer della catena, e un puntatore hash associato ad ogni buffer punta al buffer successivo nella catena. Pertanto, per tutti i riferimenti, del tipo (numero del dispositivo, numero del blocco), ad una stessa entry nella tabella hash, se il blocco corrispondente si trova nella cache di buffer, allora il buffer si troverà nella catena che parte da quell'entry della tabella hash; in questo modo, la lunghezza di ricerca nella cache di buffer è ridotta di un fattore dell'ordine di N, dove N è la lunghezza della tabella hash.

Per la sostituzione dei blocchi, si utilizza un algoritmo che sceglie il blocco usato meno di recente: dopo che un buffer è stato allocato ad un blocco del disco, non può essere usato da un altro blocco finché tutti gli altri buffer non siano stati usati più di recente, e la lista dei buffer liberi preserva questo ordine.

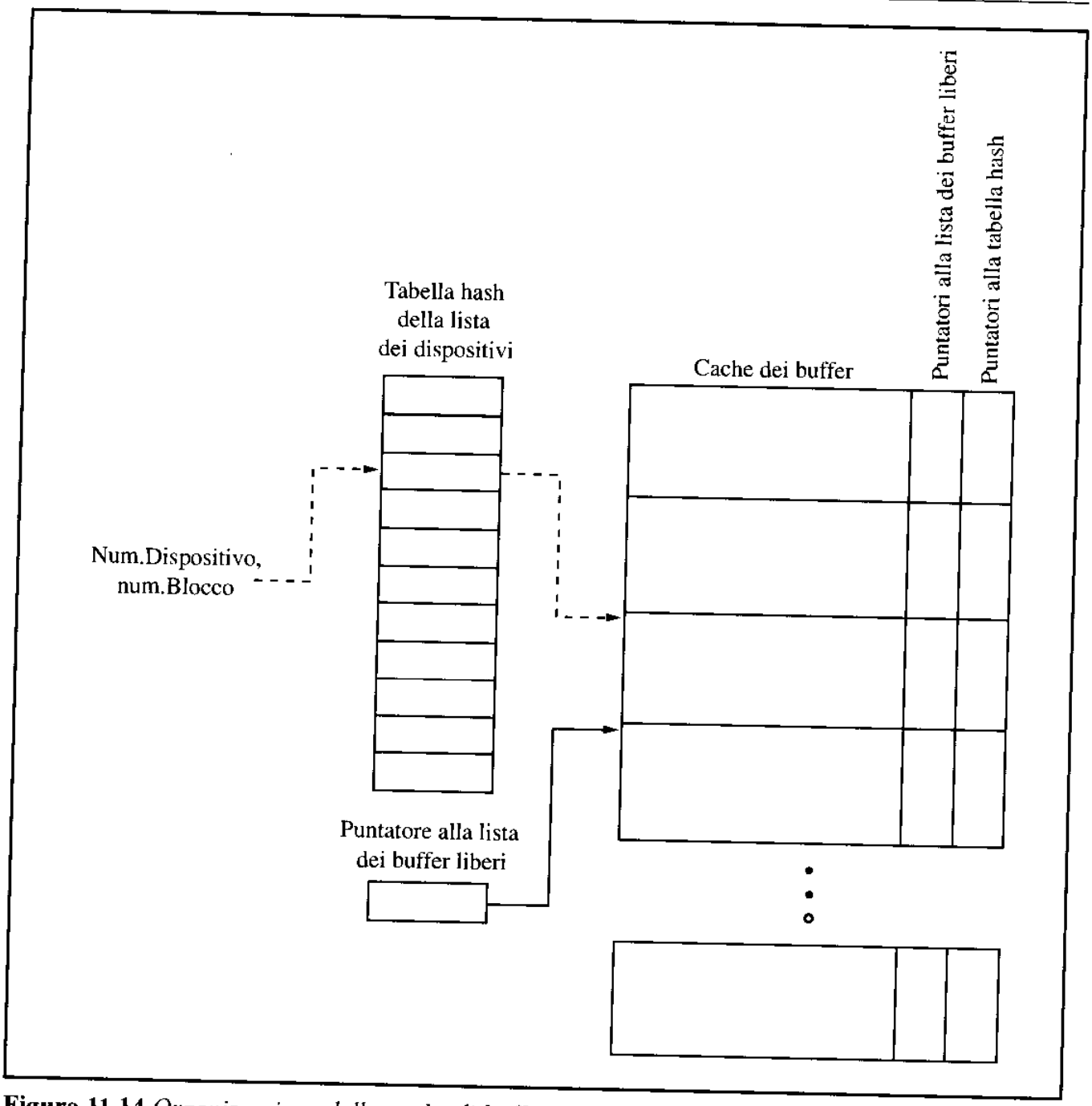


Figura 11.14 *Organizzazione della cache di buffer in UNIX*

Coda di caratteri

Dispositivi orientati a blocchi, quali dischi e nastri, possono essere serviti in modo efficiente dalla cache di buffer. I metodi di gestione di buffer adeguati ai dispositivi orientati a carattere, come i terminali e le stampanti, sono differenti: una coda di caratteri viene scritta dal dispositivo di I/O e letta dal processo, oppure viene scritta dal processo e letta dal dispositivo. In entrambi i casi, si usa il modello produttore/consumatore introdotto nel Capitolo 5. Pertanto, le code di caratteri possono essere lette solo una volta; ogni carattere che viene letto, viene effettivamente distrutto. Questo è diverso dalla cache di buffer, che può essere letta più volte e quindi segue il modello dei lettori/scrittori (anche questo descritto nel Capitolo 5).

I/O senza buffer

L'I/O senza buffer, che è semplicemente un DMA tra spazio del dispositivo e spazio del processo, è sempre il metodo più veloce che un processo ha a disposizione per effettuare l'I/O. Un processo che sta effettuando l'I/O senza buffer è bloccato in memoria e non può essere trasferito su disco; questo riduce le opportunità di effettuare trasferimenti su disco "legando" la memoria centrale, riducendo quindi le prestazioni d'insieme del sistema. Inoltre, il dispositivo di I/O è legato al processo per la durata del trasferimento, rendendolo inaccessibile ad altri processi.

Dispositivi UNIX

UNIX riconosce cinque tipi di dispositivi:

- Drive del disco
- Drive di nastro
- Terminali
- Linee di comunicazione
- Stampanti.

La Tabella 11.6 mostra i tipi di I/O che si adattano ad ogni tipo di dispositivo. I drive del disco sono pesantemente usati in UNIX, sono orientati a blocchi e potenzialmente danno un throughput piuttosto alto, quindi l'I/O per questi dispositivi tende ad essere senza buffer, o ad avvenire attraverso cache di buffer. I drive di nastro funzionano in modo simile ai drive del disco, quindi usano simili schemi di I/O.

Siccome i terminali gestiscono scambi relativamente lenti di caratteri, l'I/O di terminale tipicamente fa uso di code di caratteri e, analogamente, le linee di comunicazione richiedono l'elaborazione seriale di byte di dati per l'input o l'output, quindi sono gestiti meglio tramite code di caratteri. Infine, il tipo di I/O usato per le stampanti dipenderà in genere dalla velocità delle stesse; stampanti lente useranno code di caratteri, mentre stampanti veloci potrebbero usa-

Tabella 11.6 *Dispositivi di I/O in UNIX*

	I/O senza buffer	Cache dei buffer	Coda di caratteri
Drive del disco	X	X	
Drive del nastro	X	X	
Terminali			X
Linee di comunicazione			X
Stampanti	X		X

re I/O senza buffer; queste ultime potrebbero anche utilizzare una cache di buffer, però, siccome i dati che vanno verso una stampante non verranno mai riusati, il vantaggio fornito dalla cache di buffer non è necessario.

11.9 I/O di Windows NT

La Figura 11.15 riproduce una porzione della Figura 2.13 e mostra il gestore di I/O di Windows NT. Il gestore di I/O è responsabile di tutto l'I/O del sistema operativo, e fornisce un'interfaccia uniforme che può essere chiamata da tutti i tipi di driver.

Modelli di I/O di base

Il gestore di I/O si compone di quattro moduli:

- **Gestore di cache:** si occupa del servizio di cache per conto dell'intero sottosistema di I/O. Il gestore di cache fornisce un servizio di cache in memoria centrale a tutti i file system ed alle componenti della rete. Esso può aumentare o diminuire dinamicamente la dimensione della cache dedicata a particolari attività, man mano che l'ammontare di memoria fisica disponibile varia. Il gestore di cache contiene due servizi che servono a migliorare le prestazioni generali:
  - **Lettura pigra:** il sistema registra gli aggiornamenti nella cache e non nel disco. Più tardi, quando vi è una minore richiesta del processore, il gestore di cache scrive i cambiamenti sul disco: se un particolare blocco di cache è aggiornato nel frattempo, c'è un netto risparmio.

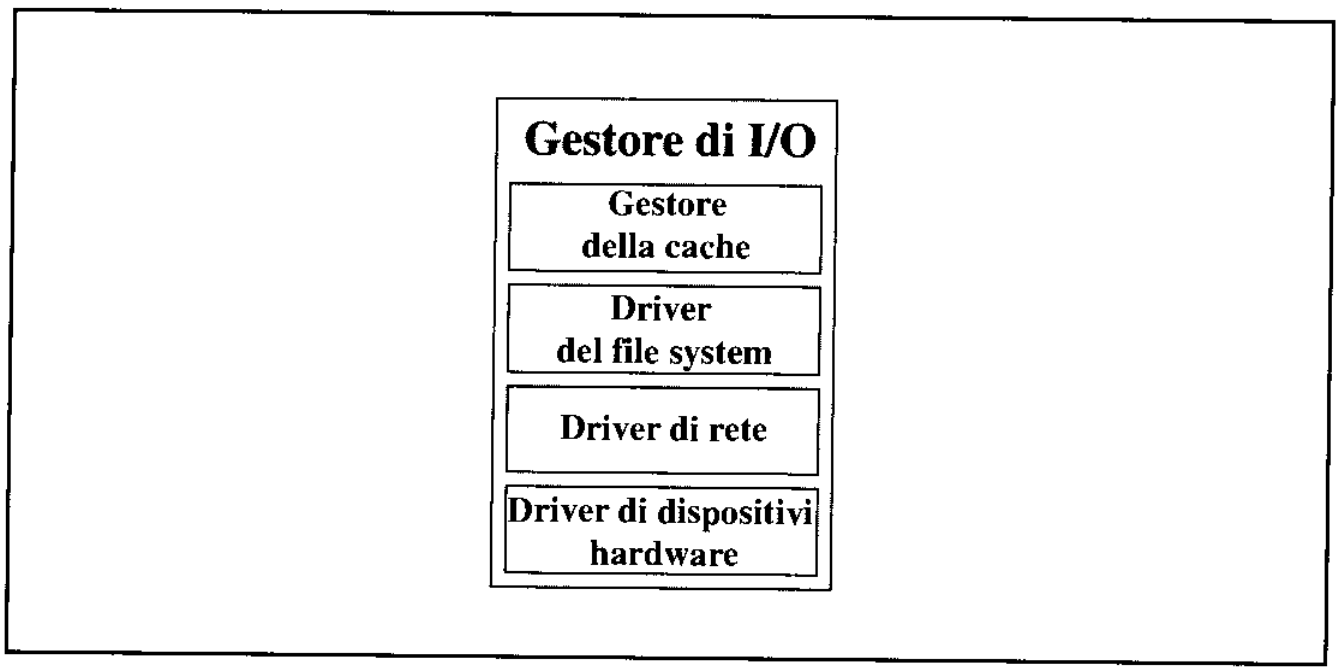


Figura 11.15 *Il gestore di I/O di Windows NT 4.0*

- **Commit pigro:** è simile alla lettura pigra per l'elaborazione di transazioni: invece di marcare immediatamente una transazione come completata con successo, il sistema mette nella cache l'informazione convalidata, e in seguito la scrive nel file di log del sistema con un processo in background.

- **Driver di file system:** il gestore di I/O tratta il driver di file system come un qualunque altro driver di dispositivo, e dirige messaggi di un certo volume ad un adeguato software di supporto per tale adattatore di dispositivo.
- **Driver di rete:** Windows NT contiene capacità di rete integrate e supporto per applicazioni distribuite.
- **Driver di dispositivi hardware:** questi driver accedono a registri hardware delle periferiche attraverso ingressi nelle librerie dinamiche di Windows NT Executive. Esiste un insieme di queste procedure per ogni piattaforma supportata da Windows NT; siccome i nomi delle procedure sono gli stessi per ogni piattaforma, il codice sorgente dei driver di dispositivo di Windows NT è portabile su diversi tipi di processori.

I/O sincrono e asincrono

L'I/O in Windows NT può operare in due diversi modi: sincrono e asincrono. Quest'ultimo è usato ogni qual volta sia possibile per ottimizzare le prestazioni delle applicazioni; con un I/O asincrono, un'applicazione inizia un'operazione di I/O e può continuare l'esecuzione mentre la richiesta di I/O viene eseguita. Con un I/O sincrono, l'applicazione è bloccata finché l'operazione di I/O non è completata.

L'I/O asincrono è più efficiente, dal punto di vista del thread chiamante, perché permette al thread di continuare l'esecuzione mentre l'operazione di I/O è messa in coda dal gestore di I/O e successivamente eseguita; comunque, l'applicazione che chiama un'operazione di I/O asincrono ha bisogno di sapere quando l'operazione è stata completata. Windows NT fornisce quattro diversi modi di segnalare il completamento di un I/O:

- **Segnalare un oggetto di tipo dispositivo di kernel:** con questo approccio, quando un'operazione relativa ad un oggetto di tipo dispositivo viene completata, si imposta un indicatore associato a quell'oggetto. Il thread che aveva richiesto l'operazione di I/O può continuare l'esecuzione finché non raggiunge un punto in cui deve fermarsi ed aspettare che l'operazione di I/O sia completata; quando l'operazione verrà completata, potrà continuare. Questa tecnica è semplice e facile da usare, ma non è adatta a gestire molte richieste di I/O. Per esempio, se un thread ha bisogno di effettuare molte azioni simultanee su un singolo file, come leggere una porzione e scrivere su un'altra porzione, con questa tecnica, il thread non potrebbe distinguere tra il completamento della lettura e il completamento della scrittura: saprebbe semplicemente che una qualche operazione di I/O richiesta su questo file è stata completata.
- **Segnalare un oggetto di tipo evento di kernel:** questa tecnica permette richieste di I/O multiple e simultanee verso un unico dispositivo o un unico file. Il thread crea un evento per ogni richiesta; in seguito, può attendere il completamento di una di queste richieste o dell'intera collezione di richieste.

- **I/O allertabile:** questa tecnica fa uso di una coda associata al thread, nota come coda di APC (chiamata di procedura asincrona, Asynchronous Procedure Call). In questo caso, il thread fa richieste di I/O, mentre il gestore di I/O mette i risultati di queste richieste nella coda APC del thread richiedente.
- **Porte di completamento di I/O:** questa tecnica è usata su Windows NT Server per ottimizzare l'uso dei thread. In sostanza, un gruppo di thread è sempre disponibile, in modo che non sia necessario creare un nuovo thread per servire una nuova richiesta.

RAID Software

Windows NT supporta configurazioni RAID, definite in [MS96] nel seguente modo:

- **RAID Hardware:** dischi fisicamente separati vengono combinati in uno o più dischi logici tramite il controller del disco.
- **RAID Software:** porzioni di spazio su disco non contigue vengono combinate in una o più partizioni logiche dal driver del disco FTDISK, che è tollerante ai guasti.

Nel RAID hardware, l'interfaccia del controller gestisce la creazione e la rigenerazione di informazioni ridondanti; nel RAID software, disponibile su NT Server, la funzionalità RAID è implementata come parte del sistema operativo, e può essere usata con un qualunque insieme di dischi multipli. Il RAID software implementa RAID 1 e RAID 5; nel primo caso (replicazione di dischi), i due dischi contenenti le partizioni primarie e replicate possono essere sullo stesso controller o su controller diversi. Quest'ultima configurazione viene chiamata disk duplexing (duplex del disco).

11.10 Sommario

L'interfaccia di un sistema di elaborazione col mondo esterno è data dalla sua architettura di I/O, che viene progettata con lo scopo di fornire mezzi sistematici per il controllo dell'interazione col mondo esterno, e di fornire al sistema operativo le informazioni necessarie per una gestione efficiente dell'attività di I/O.

La funzionalità dell'I/O è generalmente spezzata in un certo numero di livelli, con livelli inferiori che si occupano dei dettagli più vicini alle funzionalità fisiche, mentre i livelli superiori si occupano dell'I/O dal punto di vista logico; il risultato è che cambiamenti nei parametri hardware possono non riguardare la maggior parte del software di I/O.

Un aspetto chiave dell'I/O è l'uso di buffer controllati dai servizi di I/O piuttosto che da processi applicativi: il gestore dei buffer riesce a ridurre le differenze tra le velocità interne del sistema di elaborazione e le velocità dei dispositivi di I/O. L'uso di buffer separa l'effettivo trasferimento di I/O dall'area di memoria del processo applicativo, dando maggiore flessibilità al sistema operativo, nello svolgimento delle funzioni di gestione della memoria.

L'aspetto dell'I/O che ha il massimo impatto nelle prestazioni generali del sistema è l'I/O del disco; di conseguenza, a questo tipo di I/O sono stati rivolti i maggiori sforzi di ricerca e progettazione. I due approcci maggiormente usati per migliorare le prestazioni dell'I/O del disco sono la schedulazione del disco e la cache del disco.

In ogni istante, possono esserci diverse richieste di I/O in coda per lo stesso disco; il compito dello scheduler del disco è di soddisfare queste richieste, in modo da minimizzare il tempo di ricerca meccanico, e quindi migliorare le prestazioni. Per far ciò, si tiene conto della distribuzione fisica delle richieste da servire e del principio di località.

Una cache del disco è un buffer, solitamente mantenuto in memoria centrale, che funziona come una cache di blocchi del disco tra la memoria del disco e il resto della memoria centrale. Grazie al principio di località, l'uso di una cache del disco dovrebbe ridurre in modo sostanziale il numero di trasferimenti di I/O di blocchi tra la memoria centrale e il disco.

11.11 Letture raccomandate

Discussioni generali sull'I/O di un computer possono essere trovate nella maggior parte dei libri che trattano architetture di computer, quali [STAL96] e [PATT94]. Una buona discussione sulla tecnologia dei dischi si trova in [SIER90]. [WIED87] contiene un'eccellente discussione sugli aspetti di prestazioni del disco, inclusi quelli relativi alla schedulazione del disco. [CAO96] analizza la cache e lo scheduler del disco. [ROSC97] fornisce una panoramica completa di tutti i tipi di sistemi di memoria esterni, con una piccola quantità di dettagli tecnici per ognuno. Un altro buon lavoro di rassegna, che dà maggiore enfasi all'interfaccia e minore enfasi ai dispositivi in sé, è [SCHW96]. [BAKE97] fornisce una dettagliata discussione sui driver di dispositivo di Windows NT ed inoltre una buona panoramica dell'architettura dell'I/O di NT.

Una rassegna eccellente sulla tecnologia RAID, scritta dagli inventori del concetto di RAID, è [CHEN94]. Una discussione più dettagliata è pubblicata dalla RAID Advisory Board, un'associazione di fornitori e consumatori di prodotti collegati a RAID [MASS94]. [CHEN96] analizza le prestazioni di RAID. [DALT96] descrive il RAID software di Windows NT in dettaglio.

BAKE97 Baker, A. *The Windows NT Device Driver Book*. Upper Saddle River, NJ: Prentice Hall, 1997.

CAO96 Cao, P.; Felten, E.; Karlin, A.; e Li, K. "Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling" *ACM Transactions on Computer Systems*, Novembre 1996.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; e Patterson, D. "RAID: High-Performance, Reliable Secondary Storage" *ACM Computing Surveys*, Giugno 1994.

CHEN96 Chen, S., e Towsley,D."A Performance Evaluation of RAID Architectures" *IEEE Transactions on Computers*, Ottobre 1996.

DALT96 Dalton, W., et al. *Windows NT Server 4: Security, Troubleshooting, and Optimization*. Indianapolis, IN: New Riders Publishing, 1996.

MASS94 Massiglia, P. (ed.). *The RAIDbook: A Source Book for Disk Array Technology*. St. Peter, MN: The Raid Advisory Board, 1994.

PATT94 Patterson, D., e Hennessy, J. *Computer Organization and Design: The Hardware/ Software Interface*, San Mateo, CA: Morgan Kaufmann, 1994. Edizione italiana *Struttura, organizzazione e progetto dei calcolatori*, Jackson Libri, 1999.

ROSC97 Rosch, W. *The Winn L. Rosch Hardware Bible*. Indianapolis, IN: Sams, 1997.

SCHW96 Schwaderer, W., e Wilson, A. *Understanding I/O Subsystems*. Milpitas, CA: Adaptec Press, 1996.

SIER90 Sierra, H. *An Introduction to Direct Access Storage Devices*. Boston, MA: Academic Press, 1990.

STALL96 Stallings, W. *Computer Organization and Architecture*, 4th Edition. Upper Saddle River, NJ: Prentice Hall, 1996.

WIED87 Wiederhold, G. *File Organization for Database Design*. New York: McGraw-Hill, 1987.

11.12 Problemi

11.1 Effettuare lo stesso tipo di analisi della Tabella 11.3 per la sequenza di richieste di tracce: 27, 129, 110, 186, 147, 41, 10, 64, 120. Supporre che la testina sia inizialmente posizionata sulla traccia 100 e che si stia muovendo in direzione dei numeri di traccia decrescenti. Ripetere la stessa analisi, supponendo che la testina questa volta si stia muovendo in direzione dei numeri di traccia crescenti.

11.2 Considerare un disco con N tracce numerate da 0 a $(N\text{-}1)$, e supporre che i settori richiesti siano distribuiti in modo casuale ma uniforme sul disco. Vogliamo calcolare il numero medio di tracce attraversate durante una ricerca.

a. Prima di tutto calcolare la probabilità di una ricerca di lunghezza j quando la testina si trova sulla traccia t. *Suggerimento:* occorre determinare il numero totale di combinazioni, e riconoscere che tutte le tracce sono equiprobabili come potenziali destinazioni.

b. Ora calcolare la probabilità di una ricerca di lunghezza K. *Suggerimento:* questo richiede la somma su tutte le possibili combinazioni di movimenti di K tracce.

c. Calcolare il numero medio di tracce attraversate da una ricerca, usando la seguente formula del valore atteso:

$$E[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

d. Mostrare che per valori grandi di N, il numero medio di tracce attraversate da una ricerca si avvicina a $N/3$.

11.3 La seguente equazione è stata proposta, sia per la cache di memoria sia per la cache del disco:

$$T_S = T_C + M \times T_D$$

Generalizzare questa equazione ad una gerarchia di memoria a N livelli invece che 2 soli livelli.

11.4 L'algoritmo di sostituzione basata sulla frequenza (Figura 11.11), definisce F_{nuovo}, F_{medio} e $F_{vecchio}$ come le frazioni della cache che comprendono la sezione nuova, media e vecchia rispettivamente. Chiaramente, $F_{nuovo} + F_{medio} + F_{vecchio} = 1$. Caratterizzare la politica quando

a. $F_{vecchio} = 1 - F_{nuovo}$

b. $F_{vecchio} = 1/$ (dimensione della cache)

11.5 Qual è il tasso di trasferimento di un'unità di nastro magnetico a 9 tracce, la cui velocità di nastro è di 120 pollici al secondo e la cui densità è di 1600 bit per pollice?

11.6 Si consideri una bobina di nastro da 2400 piedi, con un gap tra record di 0.6 pollici, e che si ferma tra una lettura ed un'altra; si supponga che la velocità media con cui il nastro accelera/rallenta durante i gap sia lineare e che tutte le altre caratteristiche del nastro siano le stesse del Problema 11.5. I dati del nastro sono organizzati in record fisici, dove ogni record contiene un numero fisso di unità definite dall'utente, chiamati *record logici.*

a. Quanto tempo occorrerà per leggere un intero nastro di record logici di 120 byte organizzati in gruppi da 10 per ogni blocco fisico?

b. E se l'organizzazione è in 30 record logici per ogni record fisico?

c. Quanti record logici staranno sul nastro, per ciascuna delle organizzazioni sopra descritte?

d. Qual è il tasso di trasferimento effettivo relativo ai due punti precedenti?

e. Qual è la capacità del nastro?

11.7 Calcolare quanto spazio del disco (in settori, tracce e superfici) sarà richiesto per memorizzare i record logici letti nel Problema 11.6 se il disco ha settori fissi a 512 byte per settore con 96 settori per traccia, 110 tracce per superficie e 8 superfici. Ignorare le intestazioni dei file e gli indici delle tracce, e supporre che i record non possono essere divisi tra due settori.

11.8 Considerare il sistema del disco descritto nel Problema 11.7 e supporre che il disco ruoti a 360 rpm. Un processore legge un settore dal disco usando un I/O guidato da interrupt, con un interrupt per byte. Se sono necessari 2.5 μs per eseguire ogni interrupt, che percentuale di tempo il processore passerà gestendo l'I/O (indipendentemente dal tempo di ricerca)?

11.9 Ripetere il Problema 11.8 usando il DMA e supponendo un interrupt per settore.

11.10 Un computer a 32 bit ha due canali selettori e un canale multiplexor. Ogni canale selettore supporta due dischi magnetici e due unità di nastro magnetico; il multiplexor ha due stampanti in linea, due lettori di schede e dieci terminali VDT connessi ad esso. Supponendo i seguenti tassi di trasferimento:

Drive del disco	800 kilobyte/s
Drive di nastro	200 kilobyte/s
Stampanti	6.6 kilobyte/s
Lettore di schede	1.2 kilobyte/s
VDT	1 kilobyte/s

stimare il tasso di trasferimento di I/O aggregato massimo per questo sistema.

11.11. Dovrebbe essere chiaro che lo striping del disco può migliorare la percentuale di trasferimento di dati quando una fetta è piccola rispetto alla dimensione della richiesta di I/O; dovrebbe anche essere chiaro che RAID 0 fornisce migliori prestazioni di un unico disco grande, perché richieste di I/O multiple possono essere servite in parallelo. Comunque, in quest'ultimo caso, è necessario effettuare striping del disco? Ovvero, lo striping del disco migliora la percentuale di I/O in confronto ad un array di dischi analogo, ma senza striping?

11.12 Si consideri una configurazione RAID a 10 drive. Completare la seguente tabella che confronta i vari livelli RAID:

Livelli di RAID	Densità di memorizzazione	Prestazioni di banda	Prestazioni di transazione
0	1		1
1			
2			
3		1	
4			
5			

Ogni parametro è normalizzato al livello RAID che fornisce la migliore prestazioni; la densità di memorizzazione si riferisce alla porzione di disco disponibile per i dati utente, mentre le prestazioni di banda riguardano quanto velocemente i dati possono essere trasferiti al di fuori dell'array. Le prestazioni di transazione misurano quante operazioni di I/O per secondo l'array può eseguire.

Appendice 11A Dispositivi di memorizzazione a disco

Disco magnetico

Un disco è un piatto circolare fatto di metallo o di plastica, rivestito di materiale magnetico. I dati sono registrati e successivamente recuperati dal disco tramite una spira conduttrice detta testina. Durante un'operazione di lettura o scrittura, la testina è stazionaria mentre il piatto ruota sotto di essa.

Il meccanismo di scrittura è basato sul fatto che l'elettricità che scorre attraverso la spira produce un campo magnetico; vengono mandati impulsi alla testina, e si registrano sequenze magnetiche sulla superficie sottostante, con diverse sequenze a seconda che la corrente sia positiva o negativa. Il meccanismo di lettura è basato sul fatto che un campo magnetico che si muove rispetto ad una spira, produce corrente elettrica nella spira; quando la superficie del disco passa sotto la testina, genera una corrente della stessa polarità di quella precedentemente registrata.

Organizzazione dei dati e formattazione

La testina è un dispositivo relativamente piccolo capace di leggere o scrivere una porzione del piatto che ruota sotto di essa: questo porta ad organizzare i dati sul piatto in un insieme di anelli concentrici, chiamati **tracce**. Ogni traccia è della stessa ampiezza della testina; tipicamente ci sono dalle 500 alle 2000 tracce per superficie.

La Figura 11.16 mostra questa struttura dei dati. Tracce adiacenti sono separati da **gap**. Questi impediscono, o quanto meno minimizzano, gli errori dovuti al cattivo allineamento della testina, o semplicemente ad interferenze nel campo magnetico. Per semplificare l'elettronica, tipicamente si mette lo stesso numero di bit su ognuna delle tracce. Pertanto la **densità**, in bit per pollice, aumenta nello spostarsi dalla traccia più esterna alla traccia più interna (questo stesso fenomeno è presente nei dischi di un fonografo).

I dati vengono trasferiti da e verso il disco in **blocchi**. Tipicamente, il blocco è più piccolo della capacità di una traccia, di conseguenza, i dati sono memorizzati in regioni della dimensione di un blocco, chiamate **settori** (Figura 11.16). Solitamente ci sono tra i 10 e i 100 settori per ogni traccia, che possono essere di lunghezza fissa o di lunghezza variabile; per evitare di imporre requisiti di precisione esagerati al sistema, settori adiacenti sono separati da gap intertraccia (intersettore).

Occorre un qualche modo per localizzare la posizione di un settore all'interno di una traccia, cioè deve esserci un punto di inizio della traccia e un modo per identificare l'inizio e la fine di ogni settore: queste caratteristiche sono gestite attraverso dati di controllo registrati sul disco, pertanto, il disco viene formattato con alcuni dati extra che sono usati solo dal drive del disco e non sono accessibili all'utente.

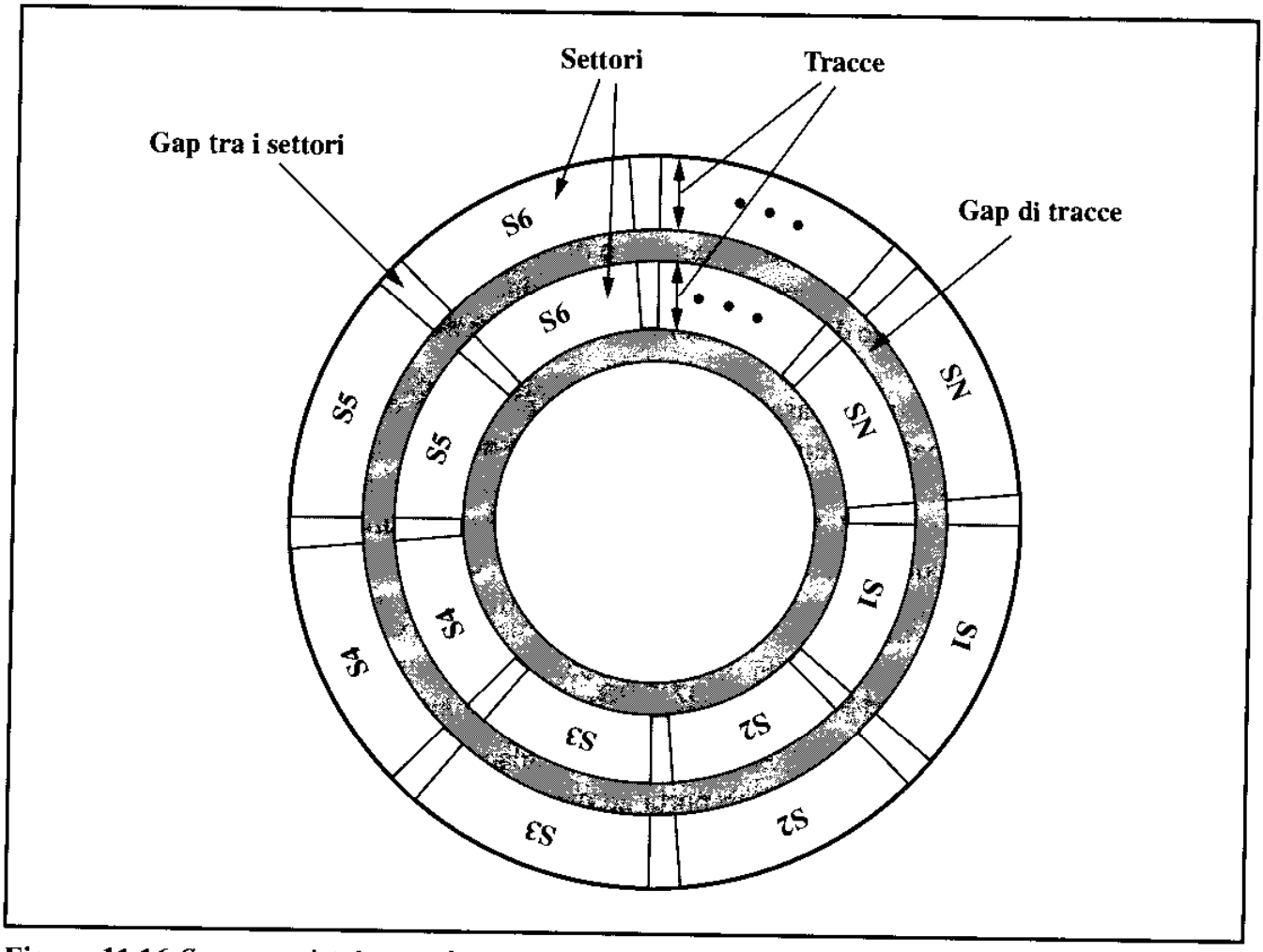


Figura 11.16 *Struttura dei dati su disco*

Caratteristiche fisiche

La Tabella 11.7 contiene le principali proprietà che caratterizzano i vari tipi di dischi magnetici. Prima di tutto la testina può essere fissa o mobile, rispetto alla direzione radiale al piatto. In

Tabella 11.7 *Caratteristiche fisiche di sistemi di dischi*

Movimento della testina	**Piatto**
Testina fissa (una per traccia)	Piatto singolo
Testina mobile (una per superficie)	Piatti multipli
Portabilià del disco	**Meccanismo della testina**
Disco non rimovibile	Contatto (floppy disk)
Disco rimovibile	Gap fissato
	Gap aerodinamico (Winchester)
Lati	
A lato singolo	
A lato doppio	

un disco a testina fissa, si ha una testina di lettura/scrittura per traccia, e tutte le testine sono montate su un braccio rigido che si stende attraverso le tracce. In un **disco a testina mobile**, c'è solo una testina di lettura/scrittura; anche in questo caso la testina è montata su un braccio e, siccome la testina deve essere in grado di posizionarsi al di sopra di ogni traccia, il braccio deve essere estensibile o ritraibile di conseguenza.

Il disco è montato in un drive del disco, che si compone di un braccio, un perno che fa ruotare il disco e l'elettronica necessaria per inserire o estrarre dati binari. Un **disco non rimovibile** è montato permanentemente sul drive del disco, mentre un **disco rimovibile** può essere rimosso e sostituito da un altro disco; questi ultimi danno il vantaggio di poter disporre di un numero illimitato di dati, con un numero limitato di dischi, e inoltre che i dischi possono essere spostati da un sistema ad un altro.

In molti dischi, la copertura magnetizzabile è stata applicata ad entrambi i lati del piatto, nel qual caso il piatto si dice a **doppio lato** (double sided), alcuni sistemi di dischi più economici usano dischi a **lato singolo** (single sided).

Alcuni drive del disco accolgono **piatti multipli** impilati verticalmente, a circa due centimetri di distanza l'uno dall'altro, con bracci multipli a disposizione; i piatti diventano quindi un'unità detta **pacchetto di dischi** (disk pack).

Infine, a seconda del meccanismo della testina, i dischi possono essere classificati in tre modi diversi: tradizionalmente, la testina di lettura/scrittura viene posizionata ad una distanza fissata al di sopra del piatto, lasciando uno spazio d'aria fra i due. All'estremo opposto possiamo avere un meccanismo della testina che produce un contatto fisico col mezzo durante un'operazione di lettura o scrittura. Questo meccanismo è usato con i **floppy disk** (dischetti), che sono piatti piccoli e flessibili, il tipo di disco più economico.

Per capire il terzo tipo del disco, bisogna discutere la relazione esistente tra densità di dati e dimensione dello spazio. La testina deve generare o captare un campo magnetico di ampiezza sufficiente ad essere scritto o letto correttamente; più stretta è la testina, più vicino deve essere il piatto per poter funzionare. Testina stretta significa tracce strette e quindi densità dei dati maggiore, come auspicabile; però più la testina è vicina al disco, maggiore sarà il rischio di errore per impurità o imperfezioni. Per spingere ulteriormente in avanti la tecnologia, è stato sviluppato il disco Winchester: le testine Winchester sono usate in drive sigillati che sono quasi privi di contaminazione, e sono progettate per operare vicino alla superficie del disco, permettendo quindi una maggiore densità dei dati. La testina è effettivamente una lamina aerodinamica posta delicatamente sulla superficie del piatto quando il disco è statico; la pressione dell'aria generata dalla rotazione del disco è sufficiente per alzare la lamina al di sopra della superficie. Il sistema risultante può essere ingegnerizzato allo scopo di usare testine più sottili che operino più vicino alla superficie delle testine rigide convenzionali.[3]

---

[3] Per interesse storico, il termine Winchester venne usato originariamente dall'IBM come nome in codice del modello del disco 3340, prima del suo annuncio. Il 3340 era un pacchetto di dischi rimovibile con le testine sigillate dentro al pacchetto; il termine è oggi applicato ad ogni unità sigillata con testina aerodinamica. Il disco Winchester si inserisce comunemente in personal computer e workstation, e viene chiamato disco fisso (hard disk).

Memoria ottica

Nel 1983 fu introdotto uno dei prodotti di maggior successo commerciale in tutti i tempi: il sistema audio digitale del compact disk (CD), un disco non cancellabile che memorizza più di 60 minuti di informazioni audio su di un lato. L'enorme successo commerciale del CD permise lo sviluppo di tecnologie di memorizzazione su disco ottico a basso costo, le quali stanno rivoluzionando la memorizzazione dei dati. Negli anni passati, è stata introdotta una varietà di sistemi di dischi ottici (Tabella 11.8); quattro di questi sistemi diventano sempre più importanti nelle applicazioni dei computer: il CD-ROM, il WORM, il disco ottico cancellabile e il DVD. Nel seguito li descriveremo brevemente.

CD-ROM

Sia il CD audio che il CD-ROM (*CD con memoria a sola lettura*, Compact Disk Read-Only Memory) condividono una tecnologia simile; la differenza principale è che i lettori di CD-ROM sono più rozzi, ed hanno dispositivi di correzione di errori che assicurano che i dati siano trasferiti correttamente dal disco al computer. Entrambi i tipi di disco sono fatti nello stesso modo, il disco è fatto da una resina, come il policarbonato, e rivestito da una superficie altamente riflettente, solitamente alluminio. L'informazione registrata in formato digitale (musica o dati) è impressa come una serie di buchi microscopici sulla superficie riflettente; questo è fatto, all'ini-

Tabella 11.8 *Dischi ottici*

CD
Compact disk. Un disco non cancellabile che immagazzina informazioni audio digitalizzate. I sistemi standard usano dischi da 12 cm e possono registrare più di 60 minuti di musica ininterrotta.

CD-ROM
Compact disk Read-Only Memory (memoria a sola lettura). Un disco non cancellabile usato per immagazzinare dati di un computer. I sistemi standard usano dischi da 12 cm che possono contenere più di 600 megabyte.

CD-I
Compact Disk Interattivo. Una specifica basata sull'uso del CD-ROM. Descrive metodi per fornire audio, video, grafica, testo e codice eseguibile su CD-ROM.

DVD
Digital Video Disk (disco video digitale). Una tecnologia che permette di produrre una rappresentazione video, digitalizzata, compressa, così come grandi volumi di dati digitali.

WORM
Write-Once Read-Many (scrivi una volta, leggi molte volte). Un disco che si può scrivere con maggiore facilità del CD-ROM, e rende commercialmente fattibile l'operazione di fare dischi in copia singola. Come col CD-ROM, dopo che è stata eseguita l'operazione di scrittura, il disco diventa a sola lettura. La dimensione più popolare è a 5.25 pollici, che può contenere da 200 a 800 megabyte di dati.

Erasable Optical Disk
Disco ottico cancellabile. È un disco che usa tecnologia ottica ma può essere facilmente cancellato e riscritto. Vengono usati sia dischi a 3.25 pollici che a 5.25 pollici. La sua capacità tipica è di 650 megabyte.

zio, con un laser ad alta intensità ben collimato, che crea il disco master. Il disco master si usa come matrice dalla quale stampare le copie; la superficie bucherellata delle copie è protetta da polvere e da graffi grazie ad una copertura superficiale di lacca trasparente.

Le informazioni contenute nel CD o nel CD-ROM vengono recuperate tramite un laser a bassa intensità posto nel lettore di dischi ottici o nell'unità drive. Il laser emette luce attraverso la copertura protettiva trasparente, mentre un motore fa girare il disco. L'intensità della luce riflessa cambia quando si incontra un buco; il cambiamento è rilevato da un fotosensore e convertito in segnale digitale.

Un buco che si trovi al centro del disco rotante passa per un punto fissato (ad esempio un raggio laser) più lentamente di un punto all'esterno, quindi occorre un modo per compensare le variazioni in velocità in modo tale che il laser possa leggere tutti i buchi alla stessa frequenza di lettura. Questo può essere fatto - come accade sui dischi magnetici - incrementando le spaziature tra i bit di informazione registrata nei segmenti del disco; in modo che l'informazione possa essere scandita ruotando il disco a velocità fissata, nota come **CAV** (*velocità angolare costante*, Constant Angular Velocity). La Figura 11.17 mostra la struttura di un disco, ottenuta usando velocità angolare costante. Il disco è diviso in un numero di settori a forma di fette di torta e in un numero di tracce concentriche. Il vantaggio di usare il sistema CAV è che i singoli blocchi di dati possono essere indirizzati tramite la traccia e il settore; per muovere la testina dalla posizione corrente ad un indirizzo specifico, serve solo un movimento corto della testina su una traccia specifica e una breve attesa che il settore cercato passi sotto la testina. Lo svantaggio di CAV è che la quantità di dati che può essere memorizzata nelle tracce esterne è la stessa che può essere messa nelle tracce interne.

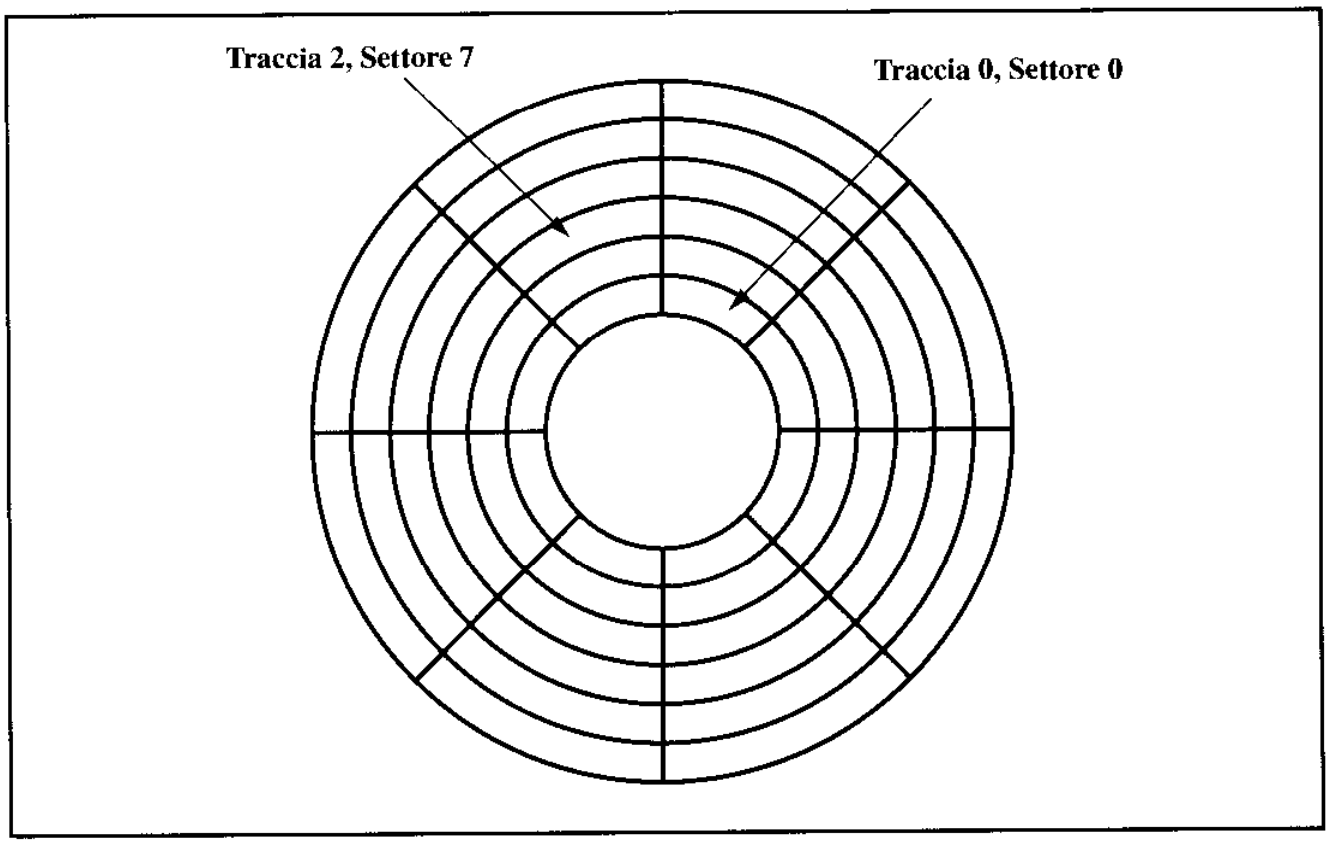


Figura 11.17 *Struttura del disco ottenuta usando velocità angolare costante (CAV)*

Siccome mettere minor informazione all'esterno del disco causa uno spreco di spazio, per CD e CD-ROM non viene usato il metodo CAV, ma l'informazione viene impacchettata uniformemente in segmenti del disco della stessa dimensione e questi vengono scanditi alla stessa frequenza, ruotando il disco a velocità variabile; i buchi vengono letti dal laser a **CLV** (*velocità lineare costante*, Constant Linear Velocity). Il disco ruota più lentamente per accessi nel pressi del bordo esterno, più velocemente per accessi nei pressi del centro. Pertanto, la capacità di una traccia e il ritardo di rotazione aumentano per le tracce che si trovano verso il bordo esterno del disco.

Sono stati prodotti CD-ROM di svariate densità; tipicamente, la spaziatura traccia a traccia potrebbe essere di 1.6 mm (1.6×10^{-6} m). L'ampiezza registrabile di un CD-ROM, lungo il suo raggio, è di 32.55 mm, cosicché il numero totale di tracce apparenti è dato dal seguente calcolo: 32550 μm diviso per la spaziatura tra tracce, quindi 20344 tracce effettive. In pratica, vi è una singola traccia a spirale, la cui lunghezza può essere calcolata moltiplicando la circonferenza media per il numero di rivoluzioni della spirale; questa risulta essere approssimativamente 5.27 km. La velocità lineare costante di un CD-ROM è 1.2 m/s, che porta ad un totale di 4391 secondi ossia 73.2 minuti, che è circa lo standard massimo di tempo di ascolto di un CD audio. Siccome i dati scorrono dal disco a 176.4 kilobyte/s, la capacità di memoria del CD-ROM è di 774.57 megabyte. Questo equivale a 550 dischetti da 3.25 pollici.

La Figura 11.18 indica la struttura usata da CD e CD-ROM. I dati sono posizionati sequenzialmente lungo una traccia a spirale. Usando la CLV, l'accesso casuale diventa più difficile. Localizzare uno specifico indirizzo richiede muovere la testina all'area generale, aggiusta-

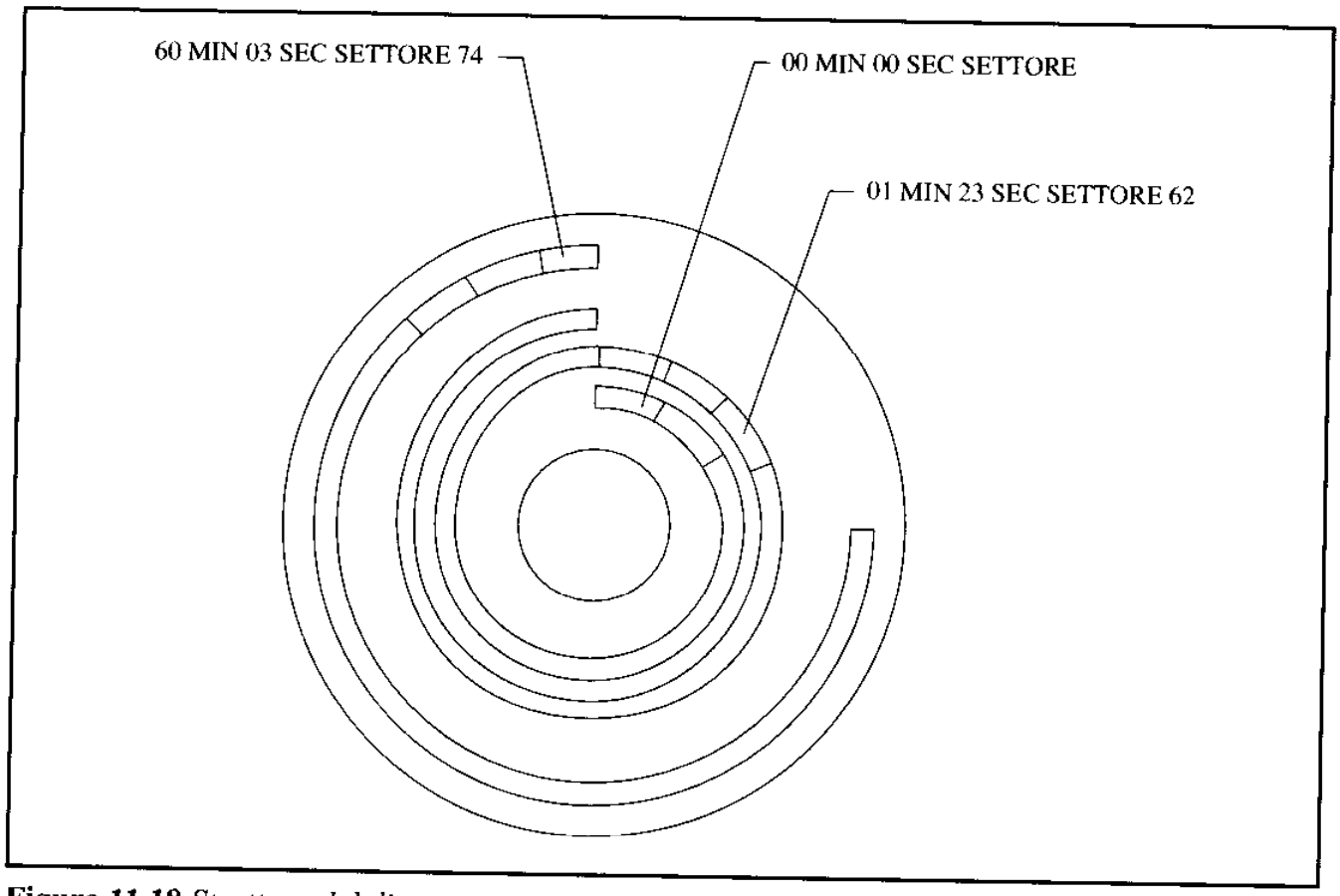


Figura 11.18 *Struttura del disco ottenuta usando velocità lineare costante (CLV)*

re la velocità di rotazione, leggere l'indirizzo, fare altri piccoli aggiustamenti per trovare il settore di interesse, ed infine accedervi.

Il CD-ROM è adeguato per la distribuzione di grandi quantità di dati ad un grande numero di utenti e, siccome il processo di scrittura iniziale è costoso, non è adatto ad applicazioni individuali. In confronto ai tradizionali dischi magnetici, il CD-ROM ha tre principali vantaggi:

- La capacità di memorizzazione dell'informazione è maggiore con i dischi ottici.
- Il contenuto di un disco ottico può essere duplicato economicamente; il contenuto di un disco magnetico viene riprodotto copiando un disco alla volta, usando due drive del disco.
- Il disco ottico è rimovibile, il che permette di usarlo come archivio. La maggior parte dei dischi magnetici non sono rimovibili, quindi, l'informazione contenuta in essi deve essere copiata su nastro prima che il drive del disco o il disco possano essere usati per memorizzare ulteriori informazioni.

Gli svantaggi dei CD-ROM sono i seguenti:

- Sono a sola lettura e non possono venire aggiornati.
- Hanno un tempo di accesso più lungo del drive del disco magnetico, fino a mezzo secondo.

WORM

Per soddisfare applicazioni in cui servono solo una o poche copie dell'insieme di dati, è stato sviluppato un CD chiamato WORM (scrivi una volta leggi più volte, Write-Once Read-Many). Il disco viene preparato in modo da poter essere scritto con un raggio laser di modesta intensità; pertanto, con un controller di CD-ROM un pochino più costoso, il cliente può scrivere il CD una volta, oltre che leggerlo. Per consentire un accesso rapido, WORM usa una velocità angolare costante, sacrificando la capacità.

Una tecnica tipica per preparare il disco è quella di usare un laser ad alta intensità che produca una serie di "vesciche" sul disco. Quando il mezzo preformattato è posto in un drive WORM, un laser a bassa intensità riesce a produrre sufficiente calore da bucare le vesciche preformate. Durante un'operazione di lettura, il laser del drive WORM, illumina la superficie del disco. Siccome le vesciche bucate producono un contrasto maggiore della superficie circostante, sono facilmente riconosciute da una semplice elettronica.

Il disco ottico WORM è adatto all'archiviazione di documenti e file, consentendo una registrazione permanente di un grande volume di dati dell'utente.

Disco ottico cancellabile

La più recente evoluzione di dischi ottici per computer è il disco ottico cancellabile, che può essere ripetutamente scritto e riscritto, come un qualunque disco magnetico. Anche se svariati altri approcci sono stati tentati, l'unica tecnologia che sembra essere commercialmente fattibile è il sistema magneto-ottico, in cui l'energia di un raggio laser viene usata insieme con un campo magnetico, per registrare e cancellare le informazioni invertendo la polarità del campo magnetico in una piccola area del disco, ricoperta da materiale magnetico. Il raggio laser scalda un'area

specifica del mezzo, e un campo magnetico può cambiare l'orientamento in quell'area mentre la temperatura è elevata; siccome il processo di polarizzazione non causa cambiamenti fisici al disco, il processo può essere ripetuto molte volte. In lettura, la direzione del campo magnetico può essere rilevata da una sorgente di luce laser polarizzata, che quando è riflessa in un particolare punto, cambierà il suo grado di rotazione a seconda dell'orientamento del campo magnetico.

Il disco ottico cancellabile ha l'ovvio vantaggio rispetto a CD-ROM e a WORM di poter essere riscritto, e quindi di essere usato come un vero supporto di memoria secondaria: cosi è competitivo col disco magnetico. I principali vantaggi del disco ottico cancellabile rispetto al disco magnetico sono i seguenti:

- **Alta capacità:** un disco ottico a 5.25 pollici può contenere 650MB di dati; i dischi Winchester più avanzati ne possono contenere meno della metà.
- **Portabilità:** il disco ottico può essere rimosso dal drive.
- **Affidabilità:** le tolleranze di ingegnerizzazione richieste ai dischi ottici sono meno severe di quelle dei dischi magnetici ad alta capacità. Pertanto, essi mostrano maggiore affidabilità e durata.

Come WORM, i dischi ottici cancellabili usano velocità angolare costante.

DVD (disco video digitale, Digital Video Disk)

Con la capacita del DVD, l'industria elettronica ha finalmente trovato un sostituto accettabile al nastro video analogico VHS: il DVD lo sostituirà nei videoregistratori (VCR) e, cosa ancora più importante in questa discussione, sostituirà il CD-ROM sui personal computer e sui server. Il DVD porta il video nell'era digitale: memorizza i film con una qualità di immagine che sorpassa i dischi laser, ed a cui si può accedere in modo casuale, come nel caso di CD audio, che possono essere suonati da un apparecchio DVD. Grandi volumi di dati possono essere infilati in un disco, al momento sette volte di più che in un CD-ROM; data l'enorme capacità di memorizzazione del DVD e la qualità dell'immagine, i videogiochi diventeranno più realistici, e il software didattico potrà incorporare più video. Grazie a questi sviluppi ci sarà un incremento di traffico su Internet e sulle intranet aziendali, via via che questo materiale verrà incluso nei siti Web.

Di seguito indichiamo alcune caratteristiche fondamentali che distinguono il DVD dal CD-ROM:

- Un DVD standard contiene 4.7GB per livello, quindi un DVD a doppio livello, lato singolo contiene 8.5GB.
- DVD usa una forma di compressione video nota come MPEG per immagini a pieno schermo di alta qualità.
- Un DVD a singolo livello può contenere due ore e 13 minuti di film, mentre un DVD a doppio livello può contenere un film lungo più di quattro ore.

CAPITOLO 12

GESTIONE DEI FILE

Nella maggior parte delle applicazioni, il file è l'elemento centrale. Qualunque sia l'obiettivo dell'applicazione, essa riguarderà la generazione e l'utilizzo d'informazioni. Con l'eccezione di applicazioni in tempo reale e alcune altre applicazioni specializzate, l'input ad un'applicazione è dato attraverso un file e, in quasi tutte le applicazioni, l'output è salvato in un file per una memorizzazione a lungo termine e per permettere accessi successivi da parte dell'utente o di altri programmi.

I file hanno una vita che prescinde dalla singola applicazione che li usa per l'input o l'output. Gli utenti vogliono essere in grado di accedere a file, salvarli e mantenere l'integrità del loro contenuto. Per favorire questi obiettivi, quasi tutti i sistemi di elaborazione forniscono sistemi distinti per la gestione di file, che solitamente consistono di programmi di utilità di sistema eseguiti come applicazioni privilegiate. Al minimo, un sistema di gestione dei file utilizza servizi speciali del sistema operativo; o addirittura, è considerato parte del sistema operativo stesso; è quindi appropriato considerare almeno gli elementi base della gestione di file all'interno di questo libro.

Inizieremo con una panoramica, seguita da una descrizione di diversi tipi di organizzazione di file. Anche se l'organizzazione di file va generalmente al di là dei compiti di un sistema operativo, è essenziale raggiungere una comprensione generale delle varie possibilità, in modo da apprezzare alcuni compromessi di progettazione che riguardano la gestione di file. La parte rimanente di questo capitolo analizza altri aspetti della gestione di file.

12.1 Introduzione

File

Quattro termini sono di uso comune quando si parla di file:

- Campo
- Record
- File
- Database.

Un **campo** è l'elemento base dei dati. Un singolo campo contiene un singolo valore, ad esempio il cognome di un impiegato, una data o il valore letto da un sensore ed è caratterizzato dalla sua lunghezza e dal tipo di dato che contiene (ad esempio, stringa ASCII, decimale). A seconda della struttura scelta per il file, i campi possono essere di lunghezza fissata o variabile e, in quest'ultimo caso, un campo solitamente è costituito da due o tre sottocampi: il valore effettivo, il nome del campo e, in alcuni casi, la lunghezza del campo. In altri casi di campi a lunghezza variabile, la lunghezza del campo è identificata usando speciali simboli di demarcazione tra campi.

Un **record** è una collezione di campi collegati che alcune applicazioni trattano come un'unità. Per esempio, il record di un impiegato, potrebbe contenere campi quali il nome, il codice fiscale, il tipo di lavoro effettuato, la data di assunzione e così via. Anche in questo caso, a seconda della struttura scelta per il file, i record possono essere di lunghezza fissata o variabile. Un record può avere lunghezza variabile o perché qualcuno dei suoi campi ha lunghezza variabile, o perché il numero dei campi è variabile; nell'ultimo caso, ogni campo è usualmente accompagnato da un nome di campo. In entrambi i casi, l'intero record solitamente contiene la lunghezza del campo.

Un **file** è una collezione di record dello stesso tipo. Il file è trattato come una singola entità dagli utenti e dalle applicazioni, e ci si può riferire ad esso tramite il suo nome. I file hanno nomi unici e possono essere creati o cancellati. Restrizioni di accesso solitamente sono applicate a livello di file, ossia, in un sistema condiviso, l'accesso all'intero file può essere concesso o negato a utenti e programmi. In alcuni sistemi più sofisticati, questi controlli possono essere estesi a livello di record o addirittura a livello di campo.

Un **database** è una collezione di dati dello stesso tipo. Tra gli aspetti essenziali di un database c'è il fatto che le relazioni esistenti tra elementi di dati sono esplicite, e che un database è progettato per l'uso da parte di diverse applicazioni.

Un database può contenere tutte le informazioni relative ad un'organizzazione o un progetto, come uno studio d'affari o scientifico; esso consiste di uno o più tipi di file. Solitamente, esiste un sistema separato per la gestione dei database, anche se questo potrebbe utilizzare alcuni dei programmi per la gestione di file.

Utenti e applicazioni vogliono usare i file. Le operazioni tipiche che devono essere fornite contengono le seguenti[LIVA90]:

- **Retrieve_all** (ritrova tutti): ritrova tutti i record di un file. Quest'operazione sarà richiesta da un'applicazione che deve elaborare tutte le informazioni contenute in un intero file. Per esempio, un'applicazione che produce un riassunto dell'informazione contenuta in un file, avrebbe bisogno di reperire tutti i record. Quest'operazione è spesso indicata col nome *elaborazione sequenziale*, in quanto si accede in sequenza a tutti i record.
- **Retrieve_one** (ritrova uno): richiede il ritrovamento di un unico record. Applicazioni orientate a transazioni, interattive, utilizzano quest'operazione.
- **Retrieve_next** (ritrova il successivo): richiede che sia ritrovato il record "successivo" a quello ritrovato più di recente, in una qualche sequenza logica. Alcune operazioni interattive, quali il riempimento di moduli, possono richiedere questa operazione. Un programma che sta effettuando una ricerca può a sua volta farne uso.
- **Retrieve_previous** (ritrova il precedente): simile all'operazione precedente, ma in questo caso si cerca il record "precedente" a quello corrente.
- **Insert_one** (inserisci uno): inserisci un nuovo record nel file. Può essere necessario che il nuovo record sia messo in una particolare posizione per mantenere la sequenzialità del file.
- **Delete_one** (cancella uno): cancella un record esistente. Alcuni link o altre strutture dati possono necessitare un aggiornamento per mantenere la sequenzialità del file.
- **Update_one** (aggiorna uno): ritrova un record, aggiorna uno o più campi e riscrivi il record aggiornato all'interno del file. Anche in questo caso, può essere necessario preservare la sequenzialità. Se la lunghezza del record è cambiata, l'operazione di aggiornamento è solitamente più difficile.
- **Retrieve_few** (ritrova alcuni): ritrova un certo numero di record. Per esempio, un'applicazione o un utente potrebbero aver bisogno di recuperare tutti i record che soddisfano un determinato insieme di criteri.

La natura delle operazioni eseguite più di frequente su un file, influenzerà il modo in cui il file è organizzato, come sarà discusso nella Sezione 12.2.

Sistemi di gestione di file

Un sistema di gestione di file è l'insieme del software di sistema che fornisce servizi agli utenti e alle applicazioni per permettere loro l'utilizzo dei file. Tipicamente, l'unico modo che utenti e applicazioni hanno a disposizione per accedere ai file è attraverso il sistema di gestione di file. Questo solleva l'utente o il programmatore dal bisogno di sviluppare software dedicato per ogni applicazione, e fornisce al sistema un mezzo per controllare le sue risorse più importanti. [GROS86] suggerisce i seguenti obiettivi, per un sistema di gestione di file:

- Incontrare i bisogni della gestione dei dati e i requisiti dell'utente, che comprendono la memorizzazione dei dati e la capacità di eseguire le operazioni precedentemente elencate.
- Garantire, il più possibile, che i dati nel file siano mantenuti validi.

- Ottimizzare le prestazioni, sia dal punto di vista del sistema in termini di throughput globale, che dal punto di vista dell'utente in termini di tempo di risposta.
- Fornire supporto di I/O per una varietà di dispositivi di memorizzazione dei dati.
- Minimizzare o eliminare la probabilità di dati persi o distrutti.
- Fornire un insieme di procedure di interfaccia all'I/O standardizzate.
- Fornire supporto all'I/O a più utenti, nel caso di sistema multiutente.

A proposito del primo punto, l'ampiezza della gamma di requisiti atti a soddisfare le richieste del cliente, dipende dalla varietà di applicazioni e dall'ambiente in cui il sistema di elaborazione è usato. Per un sistema ad utilizzo generale, interattivo, l'insieme di requisiti minimali è il seguente:

1. Ogni utente dovrebbe essere in grado di creare, cancellare e modificare i file.
2. Ogni utente può avere un accesso controllato ai file di altri utenti.
3. Ogni utente può controllare quali tipi di accesso sono permessi sui suoi file.
4. Ogni utente dovrebbe essere in grado di ristrutturare i propri file in una forma appropriata al problema.
5. Ogni utente dovrebbe essere in grado di trasferire dati tra file.
6. Ogni utente dovrebbe essere in grado di fare backup e ripristino dei file nel caso di guasto.
7. Ogni utente dovrebbe essere in grado di accedere ai propri file tramite nomi simbolici.

Questi obiettivi e requisiti dovranno essere tenuti a mente durante le nostre discussioni riguardanti i sistemi di gestione di file.

Architettura di un file system

Un modo per farsi un'idea delle possibilità della gestione dei file è quello di analizzare una rappresentazione di una tipica organizzazione del software, come quella suggerita nella Figura 12.1. Naturalmente, diversi sistemi saranno organizzati in modo diverso, ma questo schema è piuttosto comune. Al livello più basso, i **driver di dispositivo** comunicano direttamente con le periferiche o i loro controllori o canali. Un driver di dispositivo si occupa dell'inizio delle operazioni di I/O su un dispositivo e di effettuare il completamento di una richiesta di I/O. Per operazioni riguardanti file, i tipici dispositivi controllati sono i drive di disco e di nastro. I driver di dispositivo sono usualmente considerati parte del sistema operativo.

Il livello successivo è chiamato **file system base**, o livello dell'**I/O fisico**. Esso è l'interfaccia primaria di un sistema di elaborazione con l'ambiente esterno. Si occupa dei blocchi di dati che sono scambiati con i sistemi di disco o nastro ed inoltre, di posizionare questi blocchi all'interno dei dispositivi di memoria secondaria e di organizzare i buffer in memoria centrale. Esso non capisce il contenuto dei dati e nemmeno la struttura dei file. Il file system base è spesso considerato parte del sistema operativo.

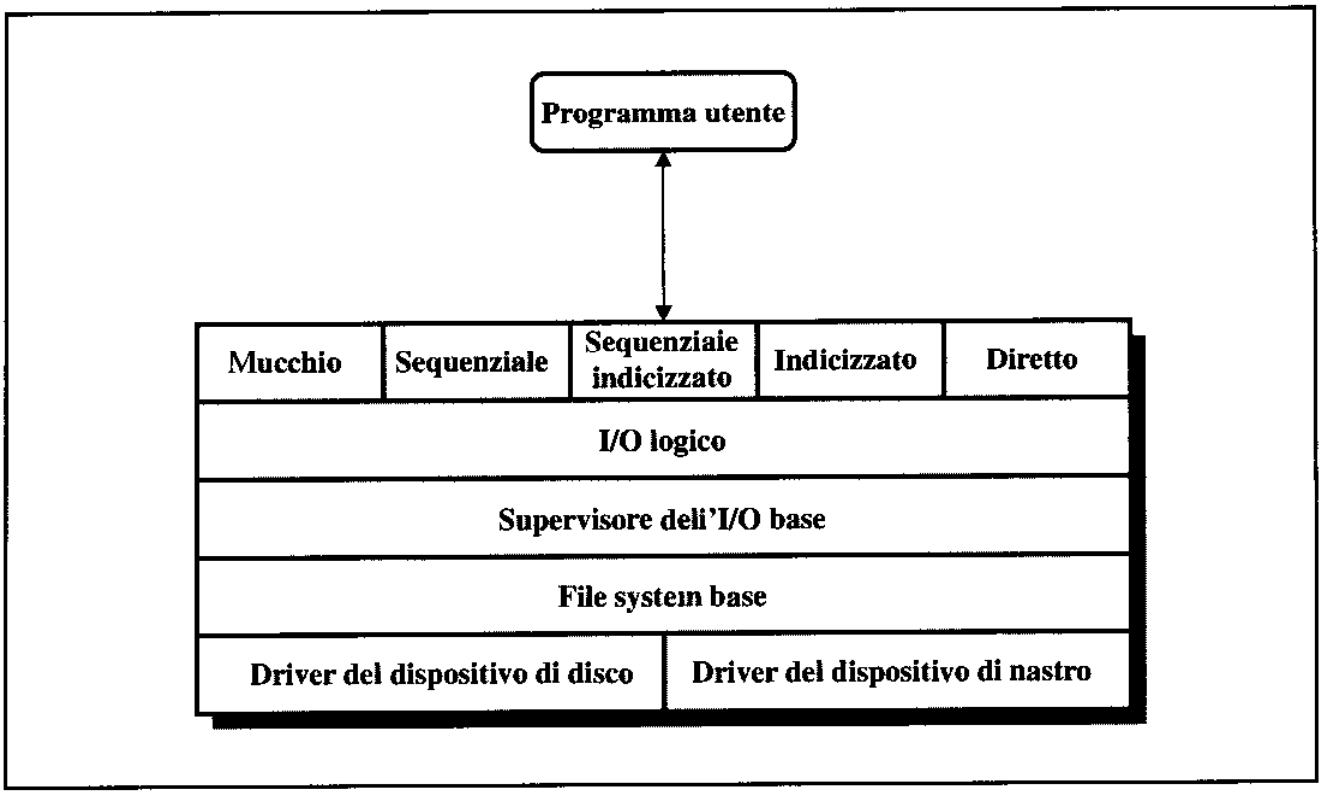


Figura 12.1 *Architettura software di un file system [GROS86]*

Il **supervisore dell'I/O base** è responsabile di tutti gli inizi e le fini dell'I/O di file. A questo livello, sono mantenute delle strutture di controllo che si occupano dei dispositivi di I/O, dello scheduling e dello stato dei file. Il supervisore dell'I/O base si occupa di scegliere il dispositivo sul quale viene espletato l'I/O di file, sulla base del file che è stato selezionato; si occupa inoltre della schedulazione del disco e degli accessi al nastro allo scopo di ottimizzarne le prestazioni. A questo livello vengono assegnati i buffer di I/O e viene allocata la memoria secondaria. Il supervisore dell'I/O di base è parte del sistema operativo.

L'**I/O logico** abilita utenti e applicazioni ad accedere ai record, quindi, mentre il file system base si occupa di blocchi di dati, il modulo di I/O logico si occupa di record di file, fornendo una funzionalità di I/O a record di tipo generale e mantenendo i dati di base riguardanti i file.

Il livello del file system solitamente più vicino all'utente è chiamato **metodo di accesso**, che fornisce un'interfaccia standard tra applicazioni e file system e i dispositivi che contengono i dati. Diversi metodi di accesso riflettono diverse strutture di file e diversi modi a cui accedere e con cui elaborare i dati. Alcuni tra i metodi di accesso più comuni sono mostrati in Figura 12.1 e sono brevemente descritti nella Sezione 12.2.

Funzioni di gestione dei file

Un altro modo di vedere le funzioni di un file system è mostrato in Figura 12.2. Seguiamo questo diagramma da sinistra a destra. Utenti e programmi applicativi interagiscono con il file system attraverso comandi per la creazione e la cancellazione di file e per l'esecuzione di operazioni sui file. Prima di eseguire una qualunque operazione, il file system deve identificare e localizzare il file selezionato; questo richiede l'uso del concetto di directory che descriva le

posizioni di tutti i file ed i loro attributi. Inoltre, la maggior parte dei sistemi condivisi rafforza il controllo degli accessi utente: solo utenti autorizzati hanno il diritto di accedere a particolari file in particolari modi. Le operazioni base che un utente o un'applicazione possono eseguire su un file, vengono eseguite a livello di record. L'utente o l'applicazione vedono il file come se avesse una qualche struttura che organizza i record, ad esempio una struttura sequenziale (ad esempio, i record del personale vengono immagazzinati in ordine alfabetico rispetto al cognome). Pertanto, per tradurre i comandi utente in comandi specifici di manipolazione file, deve essere utilizzato un metodo di accesso appropriato a questa struttura di file.

Mentre gli utenti e le applicazioni agiscono a livello record, l'I/O è fatto a livello di blocchi, pertanto i record di un file devono essere convertiti in blocchi prima di un'operazione di output e convertiti da blocchi dopo un input. Per supportare l'I/O a blocchi dei file, occorrono diverse funzioni, per esempio deve essere gestita la memoria secondaria: questo comprende allocare ai file i blocchi liberi di memoria secondaria e gestire i blocchi rimasti liberi in modo da conoscere i blocchi disponibili da usare per file nuovi o nel caso in cui quelli esistenti crescano in dimensioni. Inoltre devono essere schedulate richieste di I/O a blocchi individuali, come descritto nel Capitolo 11. Sia la schedulazione di disco che l'allocazione di file si occupano di ottimizzare le prestazioni e quindi, come era prevedibile, queste funzioni devono essere considerate insieme. Inoltre l'ottimizzazione dipenderà dalla struttura dei file e dalle strutture di accesso, di conseguenza, lo sviluppo di un sistema di gestione dei file, ottimale dal punto di vista della prestazioni, è un compito incredibilmente complicato.

La Figura 12.2 suggerisce una divisione tra quelli che potrebbero essere considerati compiti del sistema di gestione dei file preso come sistema a sé stante e i compiti del sistema operativo, avendo come punto di intersezione l'elaborazione di record. Questa divisione è arbitraria; in diversi sistemi sono stati scelti approcci diversi.

Nella parte rimanente di questo capitolo, analizzeremo alcuni aspetti di progettazione suggeriti dalla Figura 12.2. Inizieremo discutendo l'organizzazione dei file e i metodi di accesso. Anche se questo argomento va al di là di quelli che vengono usualmente definiti i compiti di un sistema operativo, è impossibile introdurre altri aspetti di progettazione legati ai file senza un'infarinatura di organizzazione di file e accesso a file. In seguito, analizzeremo il concetto di directory di file, che sono solitamente gestite dal sistema operativo a nome del sistema di gestione dei file. Gli ultimi argomenti trattati, riguarderanno aspetti fisici di I/O nella gestione dei file e verranno trattati come aspetti di progettazione di sistemi operativi; uno degli aspetti è il modo in cui record logici vengono organizzati in blocchi fisici. Infine, considereremo aspetti collegati all'allocazione di file in memoria secondaria e alla gestione di memoria secondaria libera.

12.2 Organizzazione ed accesso a file

In questa sezione usiamo il termine *organizzazione di file* nel riferirci alla struttura logica a record, così come è determinata dal modo in cui si accede ai file stessi. L'organizzazione fisica in memoria secondaria dipende dalla strategia di gestione dei blocchi e dalla strategia di allocazione, argomenti trattati più avanti in questo capitolo.

Nello scegliere l'organizzazione dei file, vi sono molti importanti criteri:

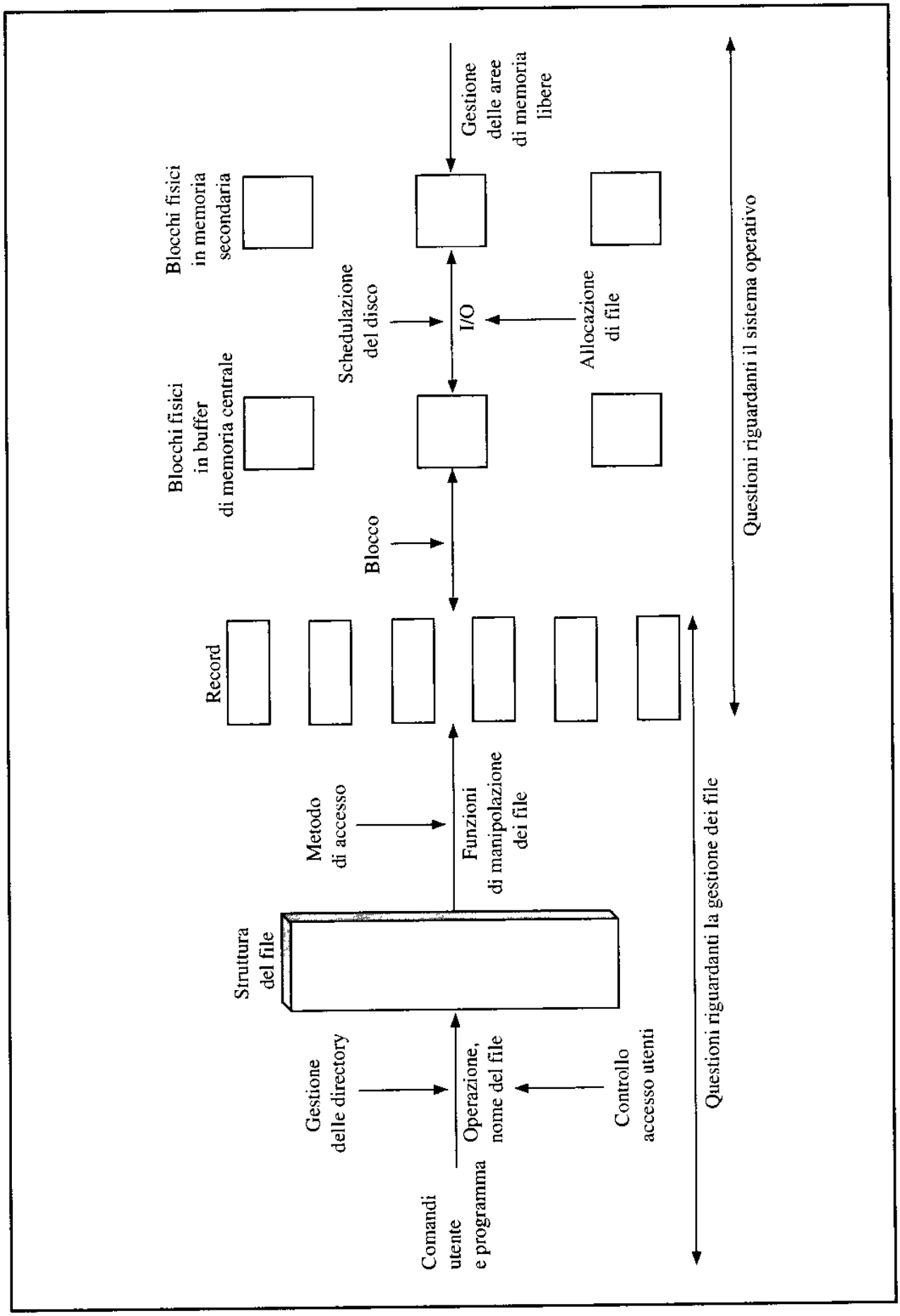


Figura 12.2 *Elementi della gestione dei file*

- Accesso rapido
- Aggiornamento semplice
- Memorizzazione economica
- Semplicità di manutenzione
- Affidabilità.

La priorità relativa di questi criteri dipende dall'applicazione che utilizza il file. Per esempio, se un file viene solo elaborato in modo batch, con accessi continui a tutti i record, allora un accesso rapido per il recupero di un singolo record è di scarsa importanza; un file mantenuto su un CD-ROM non verrà mai aggiornato, per cui la semplicità di aggiornamento non è importante.

Questi criteri possono essere in conflitto tra loro. Per esempio, per una memorizzazione economica, dovrebbe esserci una ridondanza di dati minima, ma d'altro canto, la ridondanza è il mezzo principale per l'aumento della velocità di accesso a dati. Un esempio di ciò è l'uso di indici.

Il numero di organizzazioni di file possibili che è stato implementato o anche solo proposto, è incredibilmente grande, anche per un libro che fosse unicamente dedicato ai file system. In questa breve rassegna, descriveremo cinque fondamentali configurazioni. La maggior parte delle strutture usate in sistemi reali, o ricadono in una di queste categorie, o possono essere implementate con una combinazione di esse. Le cinque organizzazioni, schematizzate in Figura 12.3, sono le seguenti:

- Mucchio
- File sequenziale
- File sequenziale indicizzato
- File indicizzato
- File diretto o a hash.

La Tabella 12.1 riassume le prestazioni relative a queste cinque organizzazioni.

Il file a mucchio

La forma di organizzazione di file meno complicata è chiamata mucchio. I dati sono collezionati nell'ordine in cui arrivano e ogni record consiste di un insieme di dati. Lo scopo di questo mucchio è semplicemente quello di accumulare la massa di dati e salvarla. I record possono avere diversi campi o campi simili ma in ordine diverso. Pertanto, ogni campo dovrebbe autodescriversi e contenere, oltre al valore, un nome di campo. La lunghezza di ogni campo può essere indicata implicitamente da delimitatori, esplicitamente inclusa come sottocampo, o nota a priori.

Siccome il file a mucchio non ha struttura, l'accesso a record avviene tramite ricerca completa, ossia, se vogliamo trovare un record contenente un particolare campo con un particolare

Tabella 12.1 *Prestazioni delle cinque organizzazioni di file principali [WIED87]*

	Spazio		Aggiornamento		Ricerca		
	Attributi		Dimensione del record		Record singolo	Sottoinsieme	Esaustivo
	Variabile	Fisso	Uguale	Maggiore			
Mucchio	A	B	A	E	E	D	B
Sequenziale	F	A	D	F	F	D	A
Sequenziale indicizzato	F	B	B	D	B	D	B
Indicizzato	B	C	C	C	A	B	D
Diretto	F	B	B	F	B	F	E

A = Eccellente, adatto allo scopo $\approx O(r)$
B = Buono $\approx O(o \times r)$
C = Adeguato $\approx O(r \log n)$
D = Richiede sforzo ulteriore $\approx O(n)$
E = Possibile con molti sforzi $\approx O(r \times n)$
F = Non adatto allo scopo $\approx O(n^{-1})$

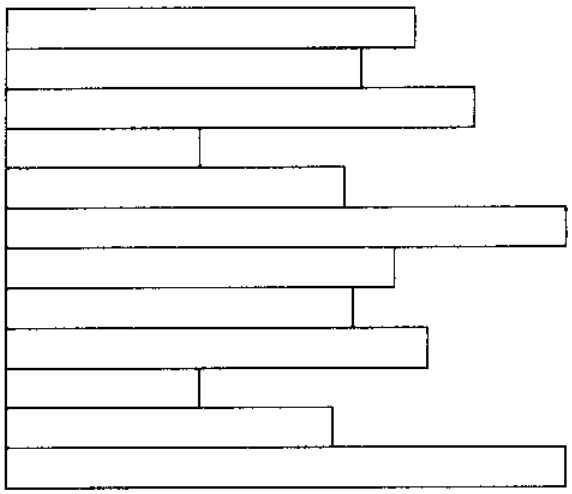
r = Dimensione del risultato
o = Numero di record nell'overflow
n = Numero di record nel file

valore, è necessario che esaminiamo ogni record nel mucchio finché non troviamo quello desiderato o finché non abbiamo analizzato l'intero file; se vogliamo cercare tutti i record che contengono un particolare campo o contengono un particolare valore in un dato campo, allora dovremo per forza analizzare tutto il file.

Si trovano file a mucchio quando i dati sono collezionati e immagazzinati prima di un'elaborazione oppure quando sono difficili da organizzare. Questo tipo di file usa bene lo spazio, nel caso in cui i dati immagazzinati varino molto in dimensione e struttura, è perfettamente adatto per ricerche complete ed è molto facile da aggiornare. Comunque, al di là di questi usi limitati, questo tipo di file è inadatto alla maggior parte delle applicazioni.

Il file sequenziale

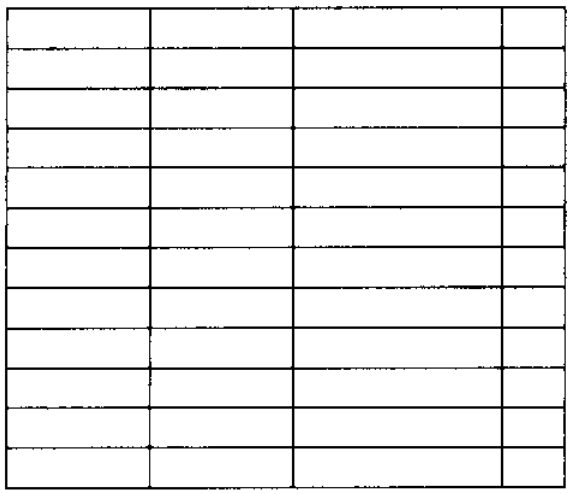
La forma più comune di struttura di file è quella del file sequenziale. In questo tipo di file, i record vengono organizzati in un formato fisso, hanno tutti la stessa lunghezza e sono costituiti dallo stesso numero di campi di lunghezza e ordine fissati. Siccome la lunghezza e la posizione dei campi è nota, occorre immagazzinare solo i valori dei campi; il nome del campo e la lunghezza di ogni campo sono attributi della struttura del file.



Record a lunghezza variabile
Insieme dei campi variabile
Ordine cronologico

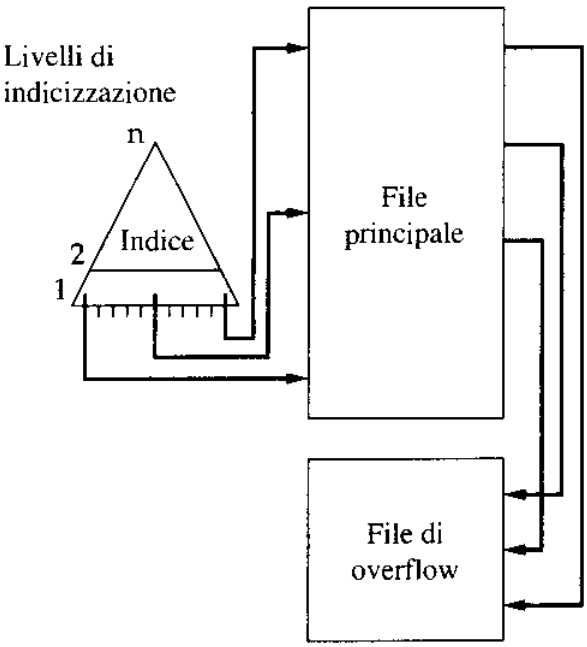
(a) File a mucchio

Campo Chiave

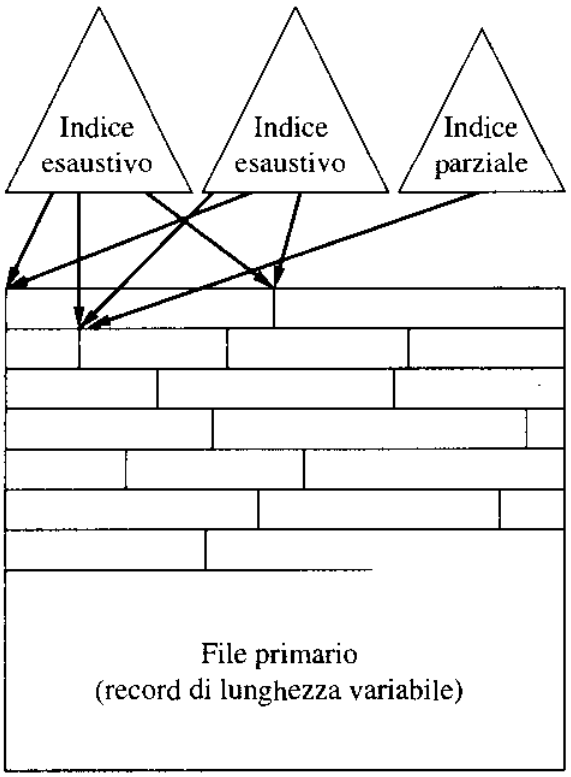


Record a lunghezza fissa
Insieme fissato di campi in un ordine fissato
Ordine sequenziale basato sul campo chiave

(b) File sequenziale



(c) File sequenziale indicizzato



(d) File indicizzato

Figura 12.3 *Comuni organizzazioni di file*

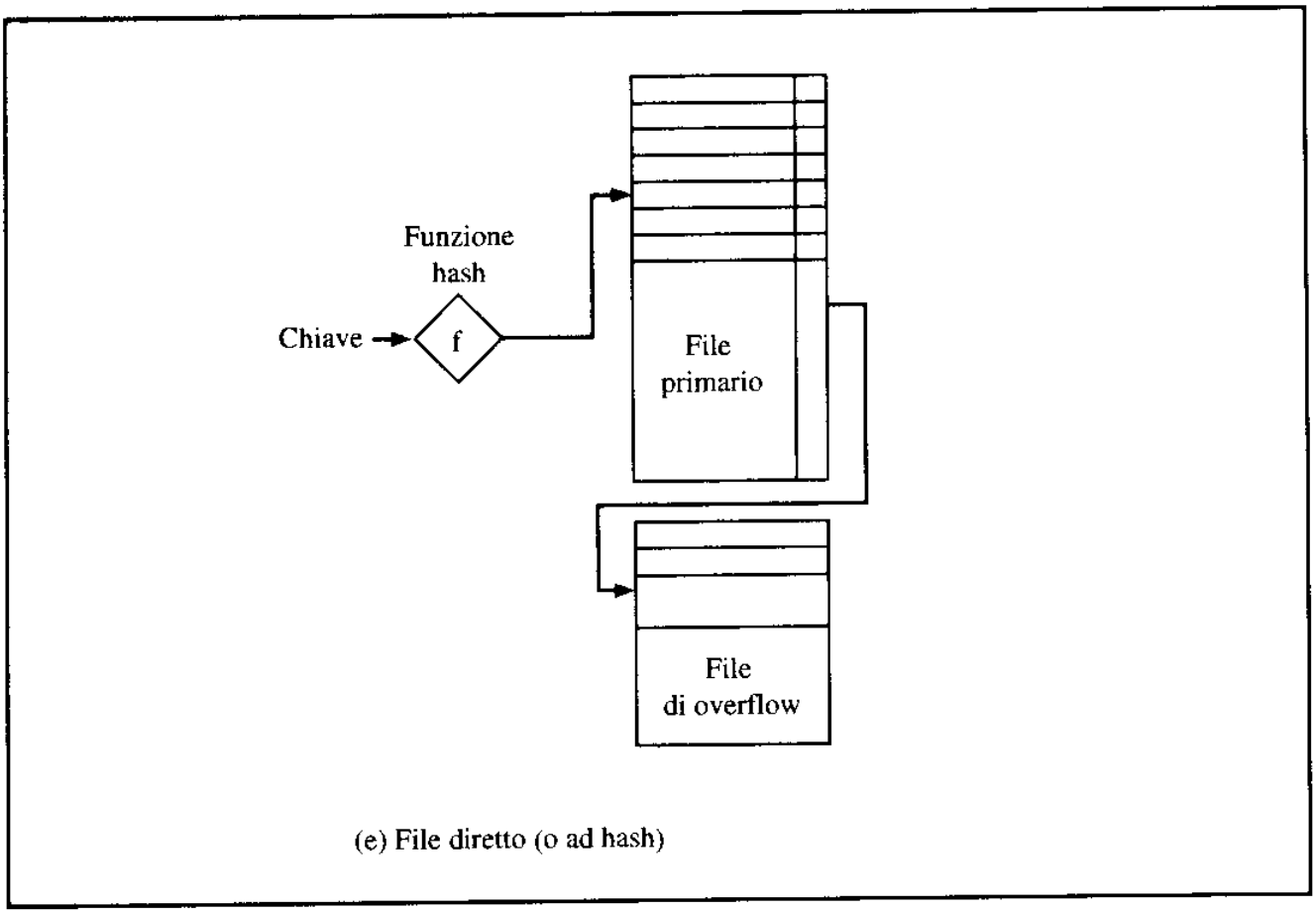


Figura 12.3 (continua) *Comuni organizzazioni di file*

Un campo particolare, solitamente il primo di ogni record, viene chiamato **campo chiave**, e identifica il record in modo unico; quindi valori chiave per record diversi sono sempre diversi. Inoltre, i record sono mantenuti in ordine di chiave: ordine alfabetico per chiavi di testo e ordine numerico per chiavi numeriche.

I file sequenziali sono tipicamente usati per applicazioni batch e sono certamente ottimi per questo tipo di applicazioni se queste richiedono elaborazione di tutti i record (ad esempio applicazioni di calcolo delle buste paga). Il file sequenziale è l'unico ad essere facilmente immagazzinato su nastro come su disco.

Per applicazioni interattive che comprendano interrogazioni e/o aggiornamenti di record individuali, il file sequenziale fornisce delle prestazioni scarse. Gli accessi richiedono ricerca sequenziale lungo il file alla ricerca della chiave voluta. Se l'intero file, o una grande fetta di esso, può essere portato in memoria centrale tutto in una volta, si possono usare tecniche di ricerca più efficienti tuttavia, accedendo ad un record in un file sequenziale, occorre una notevole quantità di elaborazione e si incontrano ritardi. Altri problemi si incontrano nel caso si vogliano effettuare aggiunte al file; un file sequenziale è solitamente immagazzinato in un semplice ordine sequenziale di record all'interno dei blocchi, ossia, l'organizzazione fisica del file su nastro o disco corrisponde esattamente all'organizzazione logica del file. In questo caso, la procedura usuale è quella di mettere i nuovi record in un file mucchio separato, chiamato log file o file delle transazioni. Periodicamente, si effettua un aggiornamento che infila il log file nel file master e produce un nuovo file con la corretta sequenza di chiave.

Un'alternativa consiste nell'organizzare fisicamente il file sequenziale come una lista collegata; in ogni blocco fisico sono immagazzinati uno o più record, ogni blocco sul disco contiene un puntatore al blocco successivo; l'inserimento di un nuovo record richiede una manipolazione di puntatori, ma non richiede che il nuovo record occupi una particolare posizione fisica nel blocco, pertanto l'organizzazione diventa più conveniente, al prezzo di elaborazioni e costi ulteriori.

Il file sequenziale indicizzato

Un approccio tipico che aggira i problemi del file sequenziale è il file sequenziale indicizzato, che mantiene la caratteristica principale dei file sequenziali: i record sono organizzati in sequenze basate su un campo chiave. Vengono aggiunte due caratteristiche: un indice al file che supporti l'accesso casuale e un file di overflow. L'indice fornisce una funzionalità di ricerca che permette di arrivare velocemente nelle vicinanze del record a cui si è interessati. Il file di overflow è simile al log file usato col file sequenziale, ma integrato in modo tale che i record appartenenti al file di overflow vengano localizzati seguendo un puntatore che parte dal loro record predecessore.

Nella struttura ad indice più semplice, viene utilizzato un singolo livello di indicizzazione. L'indice in questo caso è un semplice file sequenziale e ogni record del file indice consiste di due campi: un campo chiave, che è lo stesso del file principale, e un puntatore al file principale. Per trovare un campo specifico, si fa una ricerca per indice e si cerca il valore di chiave più grande che sia uguale o immediatamente precedente al valore chiave voluto. La ricerca continua nel file principale alla locazione indicata dal puntatore.

Per capire l'utilità di questo approccio, consideriamo un file sequenziale costituito da 1 milione di record; la ricerca di un particolare valore chiave, richiederà in media mezzo milione di accessi a record. Supponiamo ora di avere un indice contenente 1000 ingressi, dove le chiavi dell'indice siano più o meno distribuite uniformemente rispetto ai record del file principale; adesso accedere ad un particolare valore chiave richiederà in media 500 accessi al file di indice seguiti da 500 accessi al file principale per cercare il record voluto, riducendo la lunghezza media di ricerca da 500 000 a 1000 accessi.

Le aggiunte al file sono gestite nel seguente modo: ogni record del file principale contiene un campo aggiuntivo non visibile all'applicazione, che è un puntatore al file di overflow. Quando un record nuovo deve essere aggiunto al file, viene aggiunto nel file di overflow. Nel record appartenente al file principale che in sequenza logica precede il nuovo record, viene inserito un puntatore al nuovo record nel file di overflow. Se il record immediatamente precedente si trova a sua volta nel file di overflow, allora si aggiorna il suo puntatore. Come accadeva nella gestione del file sequenziale, anche in questo caso vengono effettuate ogni tanto fusioni in batch del file principale e del file di overflow.

Il file sequenziale indicizzato riduce enormemente il tempo richiesto per accedere ad un singolo record, senza sacrificare la natura sequenziale del file. Se si vuole elaborare tutto il file in modo sequenziale, si seguono sequenzialmente i record del file principale, finché non si trova un puntatore al file di overflow. A quel punto gli accessi continuano nel file di overflow, finché non si incontra un puntatore nullo, quindi si ritorna al file principale al punto in cui si era arrivati.

Per fornire una ancor maggiore efficienza negli accessi, si possono usare livelli multipli di indicizzazione, ossia, il livello inferiore di file di indice viene trattato come un file sequenziale e lo si indicizza con un file di indice di livello superiore. Consideriamo ancora un file con 1 milione di record, e un file di indice di livello inferiore con 10 000 ingressi. A questo punto si può costruire un file di indice di livello superiore con 100 ingressi. La ricerca inizia ad un livello di indicizzazione superiore (con una lunghezza di accesso media di 50 accessi) per trovare il punto di ingresso al livello di indicizzazione inferiore. A questo punto si cerca l'indice che è il punto di ingresso al file principale (lunghezza media = 50 accessi). Infine si cerca nel file principale (lunghezza media = 50 accessi). Quindi, la lunghezza di ricerca media è stata ridotta da 500 000 a 1000 e poi a 150.

Il file indicizzato

Il file sequenziale indicizzato mantiene una limitazione che era propria del file sequenziale: l'elaborazione efficiente è limitata ad un singolo campo del file. Quando è necessario cercare un record in base ad attributi diversi dal campo chiave, entrambe le forme di file sequenziali sono inadeguate e in alcune applicazioni, questo tipo di flessibilità è desiderabile.

Per ottenere questa flessibilità, occorre una struttura che usi indici multipli, uno per ogni campo su cui si potrebbe voler effettuare una ricerca. Nel file indicizzato generico, i concetti di sequenzialità e di chiave singola sono abbandonati, si accede ai record solo attraverso i loro indici. Come risultato, quindi, non vi sono limitazioni nel posizionare i record, sempre che uno degli indici contenga un puntatore che si riferisca a questo record. In più, si possono utilizzare record a lunghezza variabile.

Si usano due tipi di indici; un indice totale organizzato come un file sequenziale per facilitare le ricerche, che contiene una entry per ogni record del file principale; un indice parziale contenente gli ingressi ai record che contengono il campo di interesse. Con record a lunghezza variabile, alcuni record non conterranno tutti i campi, e quando un nuovo record è aggiunto al file principale, tutti i file indice devono essere aggiornati.

I file indicizzati sono usati principalmente in applicazioni dove sia importante che l'informazione arrivi in tempo e dove raramente tutti i dati vengono elaborati insieme. Esempi sono sistemi di prenotazioni aeree e sistemi di controllo inventari.

Il file diretto o a hash

Il file diretto o a hash, sfrutta le capacità dei dischi di accedere direttamente a ogni blocco il cui indirizzo sia noto. Come nei file sequenziali e nei file sequenziali indicizzati, ogni record ha un campo chiave, ma in questo caso non esiste il concetto di ordinamento sequenziale.

Il file diretto usa una tecnica di hashing sul valore della chiave, come spiegato nell'Appendice 8A. La Figura 8.24b mostra il tipo di organizzazione hash con file di overflow che è tipicamente usata in un file hash.

I file diretti sono spesso usati quando sono richiesti accessi molto rapidi, quando ci sono record a lunghezza fissa e quando si accede sempre un record alla volta. Esempi sono le directory, le tabelle di prezzi, le liste di nomi, gli appuntamenti.

12.3 Le directory di file

Contenuti

Ogni sistema di gestione di file, ed ogni collezione di file, ha associata una directory di file. La directory contiene informazioni sui file, compresi gli attributi, la posizione e il proprietario. La maggior parte di queste informazioni, specialmente quelle riguardanti la memorizzazione, sono gestite dal sistema operativo. La directory è essa stessa un file, di proprietà del sistema operativo e accessibile da varie procedure di gestione dei file. Anche se alcune delle informazioni contenute nelle directory sono accessibili ad utenti e applicazioni, queste informazioni vengono generalmente fornite in modo indiretto da procedure di sistema, pertanto gli utenti non possono accedere alle directory in modo diretto, nemmeno in modalità di sola lettura.

La Tabella 12.2 dà un'idea delle informazioni tipicamente contenute nella directory per ogni file del sistema. Dal punto di vista dell'utente, la directory mappa i nomi dei file, noti agli utenti e alle applicazioni, in file veri e propri, quindi ogni ingresso relativo ad un file, contiene il nome del file. Quasi tutti i sistemi hanno a che fare con diversi tipi di file e diverse organizzazioni di file, quindi anche queste informazioni sono comprese. Una categoria importante di informazioni relative ad un file, è quella riguardante la memorizzazione, che contiene la posizione e la dimensione. In sistemi condivisi, è anche importante fornire informazioni che vengano usate per controllare l'accesso a file, tipicamente un utente è il proprietario di un file e può concedere alcuni privilegi d'accesso ad altri utenti. Infine, informazioni d'uso sono necessarie a gestire l'utilizzo del file e a registrare la sua storia.

Struttura

Il modo in cui le informazioni contenute nella Tabella 12.2 vengono memorizzate, differisce ampiamente da un sistema ad un altro. Alcune delle informazioni possono essere memorizzate in un record intestazione associato al file, che riduce l'ammontare di spazio richiesto alla directory, permette la memorizzazione di tutta o di parte della directory in memoria centrale, e quindi aumenta la velocità. Alcuni elementi chiave, naturalmente, devono stare nella directory; solitamente essi contengono il nome, l'indirizzo, la dimensione e l'organizzazione.

La forma più semplice di directory è una lista di ingressi, uno per ogni file. Questa struttura potrebbe essere rappresentata semplicemente con un file sequenziale, con il nome del file che serve da campo chiave. In alcuni sistemi primitivi singolo utente, veniva usata questa tecnica. Essa diventa però inadeguata quando più utenti condividono un sistema e anche per singoli utenti che sono proprietari di molti file.

Per capire i requisiti di una particolare struttura del file, è utile considerare i tipi di operazioni che possono essere eseguite su una directory:

- **Ricerca:** quando un utente o un'applicazione fanno riferimento ad un file, si deve fare una ricerca nella directory per trovare l'entrata che corrisponde a quel file.
- **Creare un file:** quando si crea un nuovo file, occorre aggiungere una nuova entrata alla directory.

Tabella 12.2 *Informazioni contenute in una directory*

	Informazioni base
Nome del file	Nome scelto dal creatore del file (utente o programma); deve essere unico all'interno di una data directory.
Tipo di file	Ad esempio: testo, binario, modulo di caricamento, ecc.
Organizzazione del file	Per sistemi che supportano organizzazioni diverse
	Informazioai di indirizzamento
Volume	Indica il dispositivo su cui è stato memorizzato il file
Indirizzo di partenza	Indirizzo fisico di partenza in memoria secondaria (cilindro, traccia e numero di blocco su disco)
Dimensione usata	Dimensione corrente del file in byte, parole o blocchi
Dimensione allocata	Dimensione massima del file
	Informazioni di controllo degli accessi
Proprietario	Utente a cui è assegnato il controllo del file. Il proprietario può essere in grado di garantire/negare l'accesso ad altri utenti e cambiare i privilegi
Informazioni d'accesso	Una versione semplice di questo elemento dovrebbe contenere il nome e la password di ogni utente autorizzato
Azioni permesse	Controlla le letture, le scritture, le esecuzioni e le trasmissioni sulla rete
	Informazioni d'uso
Data di creazione	Quando il file è stato messo per la prima volta nella directory
Identità del creatore	Usualmente, ma non necessariamente, è il proprietario
Data dell'ultimo accesso in lettura	Data dell'ultima lettura di un record
Identità dell'ultimo lettore	Utente che ha letto il file per ultimo
Data dell'ultima modifica	Data dell'ultimo aggiornamento, inserimento o cancellazione
Identità dell'utente che ha effettuato l'nitima modifica	Utente che ha effettuato l'ultima modifica
Data dell'ultimo backup	Data dell'ultima volta in cui si è fatto il backup del file su un supporto di memorizzazione
Uso corrente	Informazioni relative all'attività corrente sul file, come i processi che hanno aperto il file, informazioni relative ad eventuali blocchi, ed eventuali aggiornamenti in memoria centrale che non sono stati riportati su disco

- **Cancellare un file:** quando si cancella un file, l'entrata corrispondente deve essere cancellata dalla directory.
- **Listare il contenuto della directory:** tutta la directory o parte di essa possono essere richieste. Solitamente questa richiesta è fatta da un utente, e il risultato è una lista di tutti i file posseduti da quell'utente, più alcuni attributi per ogni file (ad esempio il tipo, informazioni di controllo d'accesso, informazioni d'uso).

Una semplice lista non è adatta a supportare tutte queste operazioni. Consideriamo i bisogni del singolo utente; egli può avere diversi tipi di file, compresi file di testo elaborati, file grafici, fogli di calcolo e così via. L'utente potrebbe voler organizzare i suoi file per progetto, per tipo o in qualche altro modo conveniente. Se la directory è una semplice lista sequenziale, non dà alcun aiuto nell'organizzazione dei file e forza l'utente a dover stare attento a non usare lo stesso nome per due diversi tipi di file. Il problema è anche peggiore in sistemi condivisi, dove l'unicità dei nomi diventa un problema serio. Inoltre, è difficile nascondere parti della directory agli utenti, se non c'è una chiara struttura della directory stessa.

Un primo tentativo di soluzione a questi problemi potrebbe essere uno schema a due livelli. In questo caso ci sarebbe una directory per ogni utente e una directory principale; la directory principale contiene una entry per ogni directory utente, fornendo l'indirizzo e le informazioni inerenti i controlli d'accesso. Ogni directory utente è una semplice lista dei file appartenenti a quell'utente. Questa soluzione richiede che i nomi debbano essere unici solo all'interno dell'area del singolo utente, e inoltre diventa più semplice per il file system porre restrizioni d'accesso sulle directory, però continua a non fornire agli utenti, nessun aiuto nello strutturare collezioni di file.

Un approccio più potente e flessibile, quasi universalmente adottato, è l'approccio gerarchico, o ad albero (Figura 12.4). Come prima, c'è una directory principale, o radice, che ha sotto di sé un certo numero di directory utente. Ognuna di queste directory utente, a sua volta può contenere altre directory o file; questa struttura è estesa ad ogni livello, ossia, ad ogni livello una directory può consistere di ingressi per sottodirectory e/o file.

Rimane da specificare come le directory e sottodirectory siano organizzate. L'approccio più semplice, ovviamente, è quello di inserire ogni directory in un file sequenziale; siccome una directory può contenere un grande numero di entry, tale organizzazione può portare a tempi di ricerca esageratamente lunghi, in questo caso, quindi, si preferisce una struttura a hash.

Scelta dei nomi

L'utente ha bisogno di fare riferimento ad un file tramite un nome simbolico e chiaramente, ogni file nel sistema deve avere un nome unico cosicché i riferimenti ai file siano non ambigui; d'altra parte, è inaccettabile spingere affinché gli utenti usino nomi unici, specialmente in sistemi condivisi.

L'uso di una directory strutturata ad albero minimizza la difficoltà di assegnare nomi unici. Ogni file del sistema può venire localizzato seguendo il cammino dalla directory principale o radice attraverso varie ramificazioni fino a che il file non viene raggiunto. La serie di nomi di directory, che culminano col nome del file, costituisce il **pathname** (nome del cammino) del file. Ad esempio, il file che si trova nell'angolo in basso a sinistra della Figura 12.5 ha un pathname /UserB/Word/UnitA/ABC. La slash (barra diagonale) viene utilizzata per delimitare i nomi nella sequenza; il nome della directory radice è implicito, in quanto tutti i cammini iniziano da quella directory. È perfettamente accettabile avere diversi file con lo stesso nome, sempre che abbiano pathname unici; nell'esempio in figura c'è un altro file nel sistema, chiamato ABC, ma il suo pathname è /UserB/Draw/ABC.

Anche se il pathname facilita la scelta dei nomi di file, non vogliamo che l'utente debba scrivere l'intero pathname ogni volta che ha bisogno di accedere ad un file. Solitamente un

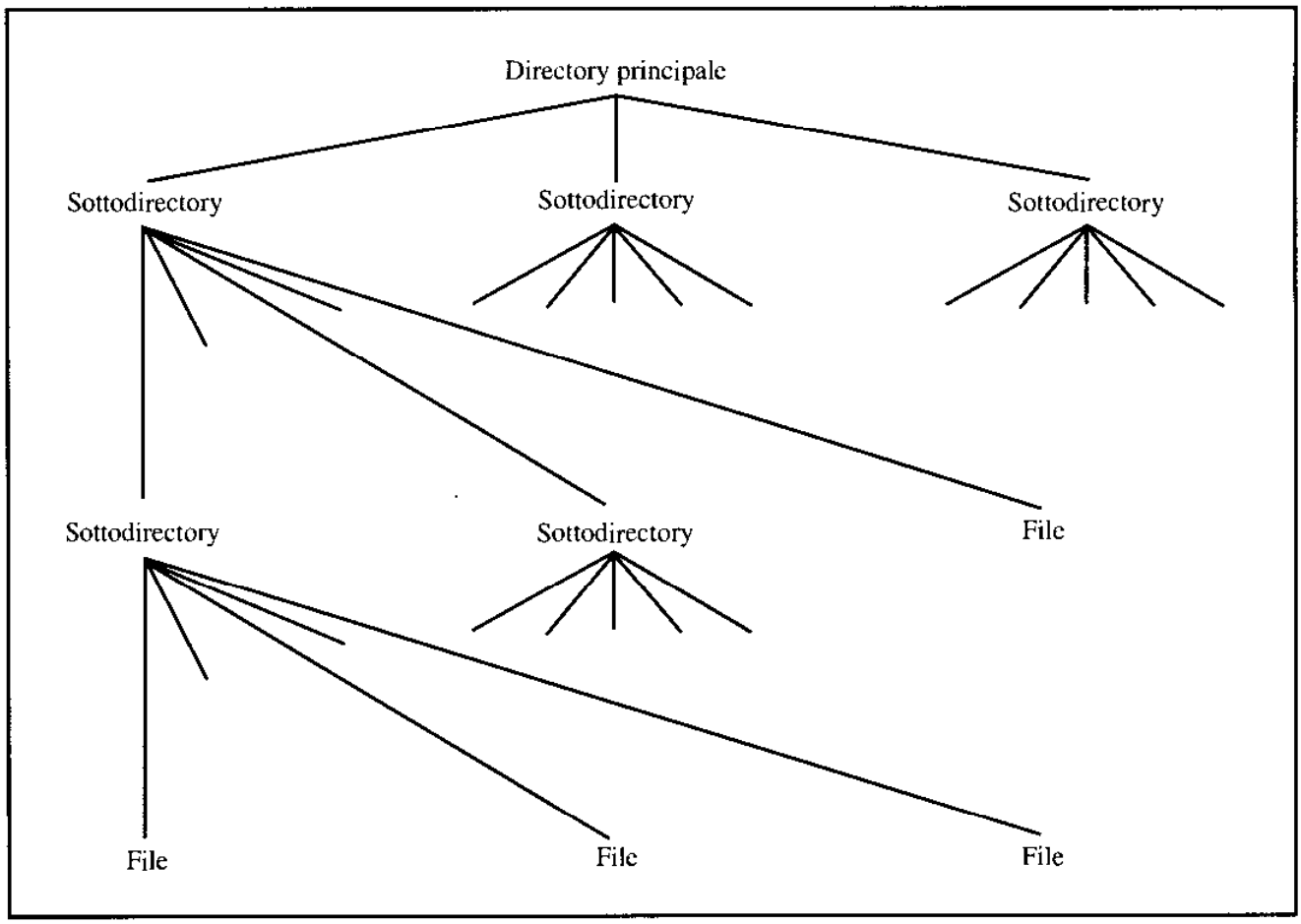


Figura 12.4 *Una directory strutturata ad albero*

utente interattivo o un processo hanno associata una directory corrente, chiamata spesso **directory di lavoro**. I file possono quindi essere chiamati in riferimento a questa directory di lavoro. Ad esempio, se la directory di lavoro per userB è "Word", il pathname UnitA/ABC sarà sufficiente ad identificare il file in basso a sinistra della Figura 12.5. Quando un utente interattivo si collega al sistema, o quando un processo è creato, la directory di lavoro di default è la directory utente. Durante l'esecuzione, l'utente può navigare su e giù per l'albero o definire una diversa directory di lavoro.

12.4 Condivisione di file

In un sistema multiutente, è quasi sempre necessario permettere che dei file vengano condivisi tra un certo numero di utenti. A questo punto sorgono due problemi: i diritti di accesso e la gestione di accessi simultanei.

Diritti di accesso

Il file system dovrebbe fornire un mezzo flessibile che permetta la condivisione di file estensiva tra utenti, ed inoltre un certo numero di opzioni che permettano di controllare il modo in cui si

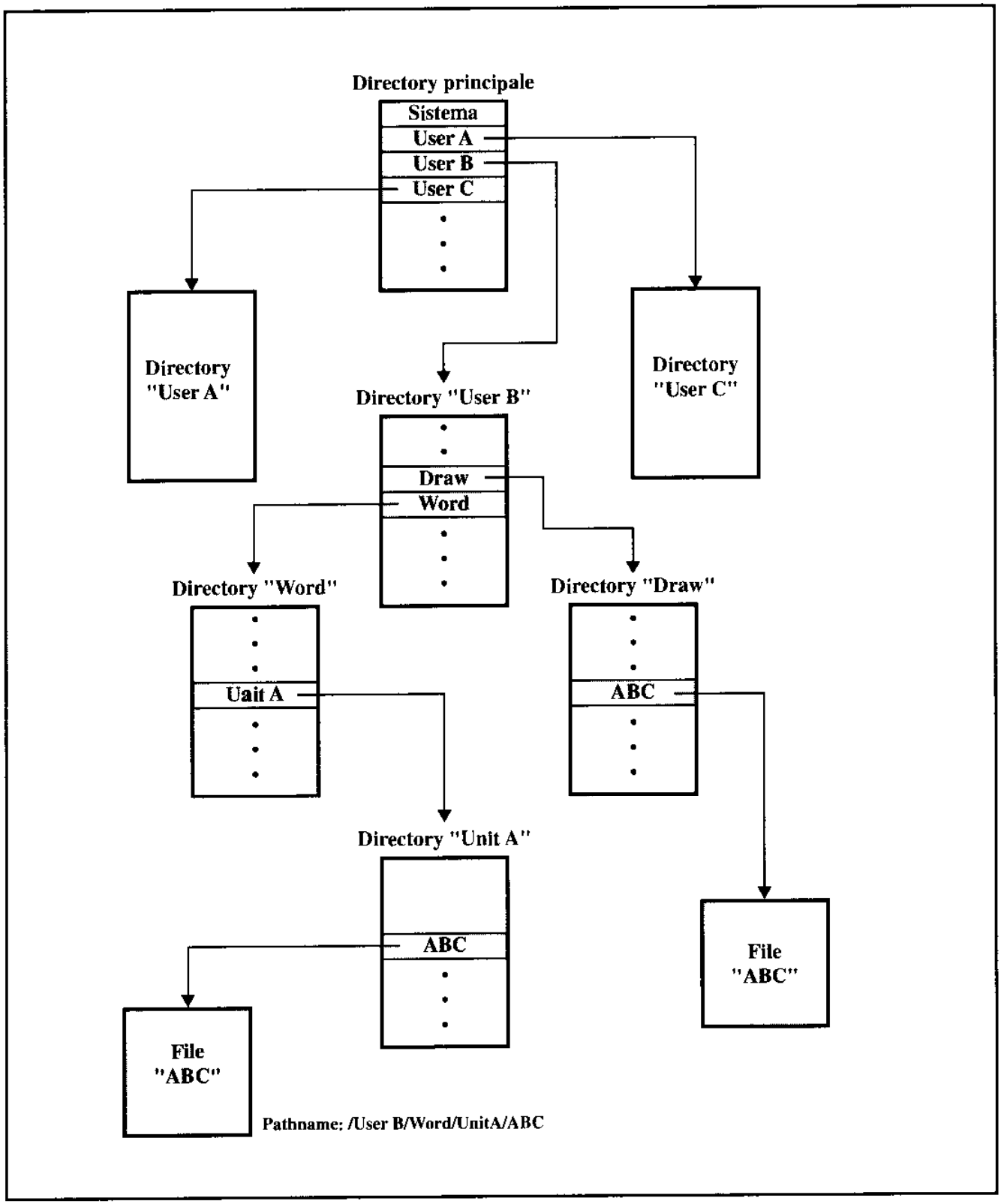


Figura 12.5 *Esempio di directory strutturata ad albero*

accede ad un particolare file. Tipicamente, utenti o gruppi di utenti hanno garantiti certi diritti di accesso a file. Si può usare una vasta gamma di diritti di accesso. La seguente lista elenca i diritti di accesso che possono venire assegnati ad un particolare utente su un particolare file:

- **Nessuno:** l'utente può non sapere dell'esistenza di un file e ancor meno accedervi. Per rafforzare questa restrizione, l'utente non dovrebbe essere in grado di leggere la directory che contiene questo file.
- **Conoscere:** l'utente può sapere che il file esiste e chi è il suo proprietario, è quindi libero di richiedere al proprietario del file gli opportuni diritti di accesso.
- **Eseguire:** l'utente può caricare ed eseguire il programma, ma non può copiarlo. Programmi proprietari sono spesso resi disponibili con queste restrizioni.
- **Leggere:** l'utente può leggere il file per qualunque scopo, ad esempio copiarlo ed eseguirlo. Alcuni sistemi sono in grado di forzare una distinzione tra vedere e copiare un file. In questo caso, il contenuto di un file può essere mostrato all'utente, ma questo non ha nessun mezzo di copiarlo.
- **Aggiungere:** l'utente può leggere e aggiungere dati al file, solitamente solo in fondo, ma non può modificare o cancellare il contenuto del file. Questo diritto di accesso è utile per collezionare dati da diverse sorgenti.
- **Aggiornare:** l'utente può modificare, cancellare e aggiungere dati al file. Questi diritti normalmente contengono scrivere il file la prima volta, riscriverlo completamente o riscrivere una parte di esso, rimuovere tutto o una porzione dei dati. Alcuni sistemi distinguono tra diversi gradi di aggiornamento.
- **Cambiare protezione:** l'utente può cambiare i diritti di accesso concessi ad altri utenti. Tipicamente, questo diritto è dato solo al proprietario del file, ma in alcuni sistemi, il proprietario può estendere ad altri questo diritto. Per evitare l'abuso del meccanismo, il proprietario del file potrà specificare quali diritti possono essere cambiati dai possessori di tali diritti.
- **Cancellare:** l'utente può cancellare il file dal file system.

Questi diritti possono essere considerati come costituenti una gerarchia, dove ogni diritto implica i diritti che lo hanno preceduto; pertanto, se un particolare utente ha il diritto di aggiornare un particolare file, avrà anche i diritti di conoscere, eseguire, leggere e aggiungere.

Un utente è scelto come proprietario di un dato file, solitamente si tratta della persona che ha creato il file stesso. Il proprietario ha tutti gli accessi descritti in precedenza e può assegnare diritti ad altri. Accessi possono essere concessi a diverse classi di utenti:

- **Utenti specifici:** utenti individuali con un certo identificatore (user ID).
- **Gruppi di utenti:** un insieme di utenti che non sono definiti in modo individuale. Il sistema deve avere un modo di tener traccia dei membri di questi gruppi.
- **Tutti:** tutti gli utenti che hanno accesso al sistema. Queste classi di accessi appartengono ai file pubblici.

Accessi simultanei

Quando più utenti possono aggiungere dati o modificare un file, il sistema operativo o il sistema di gestione dei file devono rafforzare la disciplina. Un approccio a forza bruta è quello di permettere ad un utente di bloccare l'intero file quando vuole aggiornarlo, mentre un controllo più fine consiste nel bloccare record individuali durante un aggiornamento. Essenzialmente questo è un esempio del problema dei lettori/scrittori trattato nel Capitolo 5. Aspetti di mutua esclusione e stalli devono essere affrontati nel progettare funzionalità di accesso condiviso.

12.5 Organizzazione di record a blocchi

Come indicato in Figura 12.2, i record sono unità logiche di accesso ad un file, mentre i blocchi sono le unità dell'I/O con la memoria secondaria. Durante un'operazione di I/O, i record devono essere organizzati come blocchi.

Nel fare ciò, ci sono svariati aspetti da considerare. Prima di tutto, i blocchi devono essere di lunghezza fissa o variabile? In molti sistemi, i blocchi hanno lunghezza fissa, questo semplifica l'I/O, l'allocazione di buffer in memoria centrale e l'organizzazione dei blocchi in memoria secondaria. Secondo, quale dovrebbe essere la dimensione relativa del blocco in relazione alla dimensione media del record? Il compromesso è: più grande è il blocco e più record vengono passati in un'operazione di I/O. Se un file viene elaborato, o se su di esso viene svolta una ricerca sequenziale, questo è un vantaggio, perché usando blocchi più grandi il numero di operazioni di I/O è ridotto e quindi l'elaborazione è accelerata. D'altra parte, se si accede ai record in modo casuale, senza rispettare nessuna particolare proprietà di località nei riferimenti, blocchi più grandi producono il trasferimento di record non necessari. Comunque, combinando la frequenza di operazioni sequenziali con la possibilità di riferimenti locali, possiamo affermare che il tempo di trasferimento di I/O si riduce usando blocchi più grandi. Una preoccupazione che sorge è che blocchi grandi richiedano buffer di I/O grandi, rendendo più difficile la gestione dei buffer.

Data la dimensione del blocco, ci sono tre metodi di organizzazione a blocchi che possono venire usati:

- **Organizzazione a blocchi fissa**: si usano record a dimensione fissa, e in un blocco sono inseriti un numero intero di record; può esserci dello spazio non utilizzato alla fine di ogni blocco.
- **Organizzazione a blocchi variabile con riporto**: si usano record a lunghezza variabile che vengono impacchettati nei blocchi senza spreco di spazio, cioè alcuni record possono essere messi a cavallo tra due blocchi, con la continuazione indicata da un puntatore al blocco successivo.
- **Organizzazione a blocchi variabile senza riporto**: si usano record a lunghezza variabile, ma non si usa il riporto. C'è dello spazio sprecato in quasi tutti i blocchi a causa dell'impossibilità di utilizzare la parte rimanente di un blocco se il record successivo è più grande dello spazio vuoto rimasto.

La Figura 12.6 illustra questi tre metodi, supponendo che un file sia memorizzato su disco in blocchi sequenziali. L'effetto non cambierebbe se venissero usati altri schemi di allocazione di file (si veda la Sezione 12.6).

L'organizzazione a blocchi fissa è la modalità comune per file sequenziali con record a lunghezza fissata. L'organizzazione variabile con riporto è un metodo di memorizzazione efficiente e non limita le dimensioni dei record, però è difficile da implementare; i record che si estendono su due blocchi richiedono due operazioni di I/O, e i file sono difficili da aggiornare, qualunque

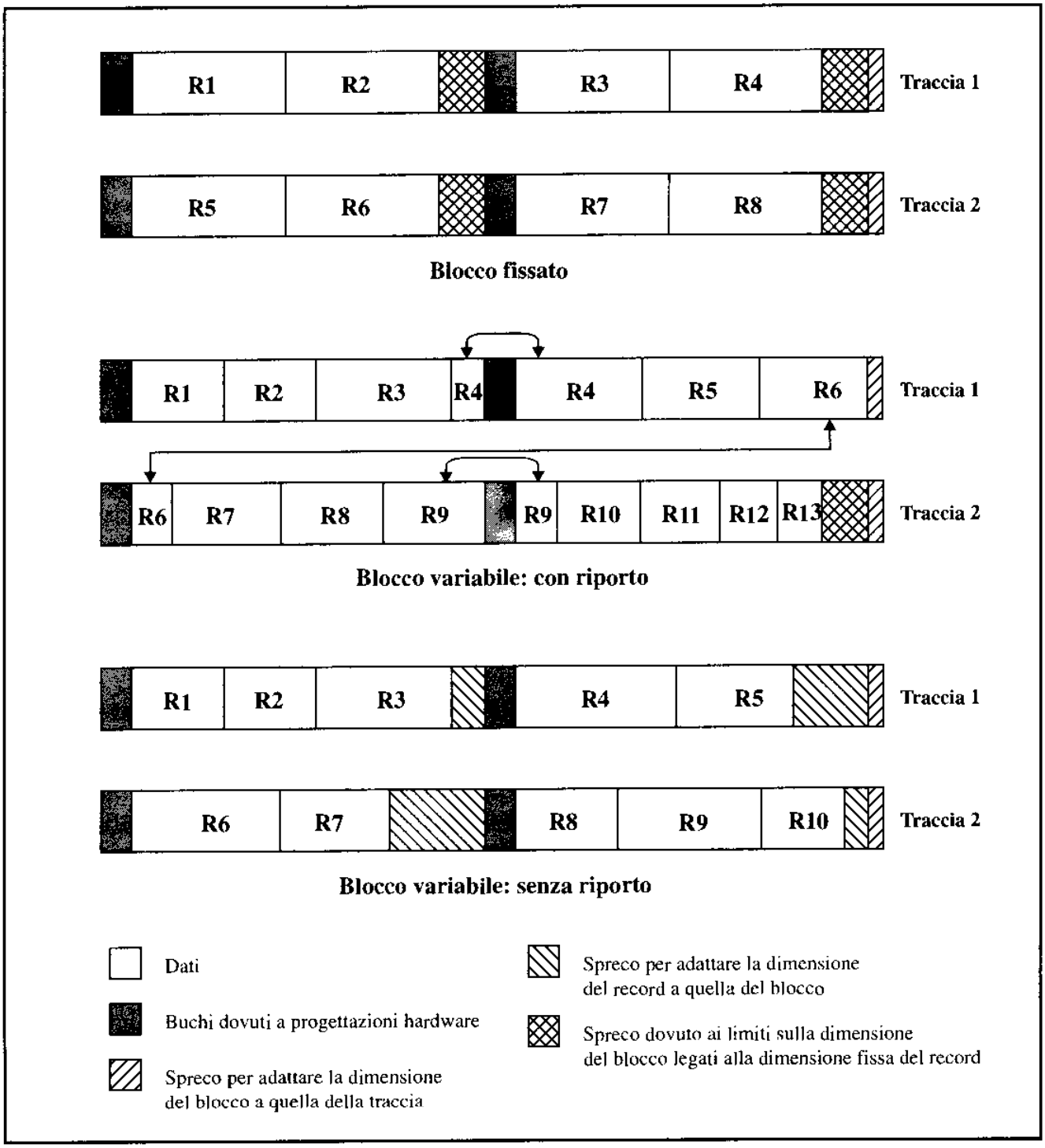


Figura 12.6 *Metodi per l'organizzazione dei record a blocchi [WIED87]*

sia l'organizzazione. L'organizzazione variabile senza riporto crea spreco di spazio e limita la dimensione di un record alla dimensione del blocco.

La tecnica di organizzazione dei record in blocchi può interagire con l'hardware di memoria virtuale, se questo è utilizzato. In un ambiente di memoria virtuale, è desiderabile scegliere la pagina come unità base di trasferimento. Le pagine solitamente sono piuttosto piccole, quindi non è pratico trattare una pagina come un blocco nel caso di organizzazione senza riporto; di conseguenza, alcuni sistemi combinano pagine multiple per creare blocchi più larghi per scopi di I/O di file. Questo approccio è usato per i file VSAM dei mainframe IBM.

12.6 Gestione della memoria secondaria

In memoria secondaria, un file consiste in una collezione di blocchi. Il sistema operativo o il sistema di gestione dei file sono responsabili dell'allocazione dei blocchi ai file. Questo solleva due problemi di gestione; prima di tutto, lo spazio in memoria secondaria deve essere allocato ai file, secondariamente, è necessario tenere traccia dello spazio disponibile per successive allocazioni. Vedremo come questi due compiti siano collegati, ossia come l'approccio scelto per l'allocazione di file possa influenzare l'approccio scelto per la gestione dello spazio libero; inoltre, vedremo che c'è un'interazione tra struttura dei file e politica di allocazione.

Iniziamo questa sezione analizzando i vari modi di allocare file su un singolo disco, in seguito affronteremo il problema di gestione dello spazio libero, ed infine discuteremo l'affidabilità.

Allocazione di file

Vi sono molti aspetti collegati all'allocazione di file:

1. Quando si crea un nuovo file, lo spazio massimo richiesto dal file viene allocato tutto in una volta?
2. Si alloca spazio per un file in una o più unità contigue, che chiameremo *porzioni*. La dimensione di una porzione può spaziare da un singolo blocco all'intero file. Quale dimensione delle porzioni dovremmo usare per l'allocazione dei file?
3. Quale tipo di struttura dati o tabella si usa per tenere traccia delle porzioni assegnate ad un file? Una tabella di questo tipo viene chiamata **FAT** (*tabella di allocazione dei file,* File Allocation Table).

Esaminiamo questi aspetti ad uno ad uno.

Confronto tra preallocazione e allocazione dinamica

In una politica di preallocazione, la dimensione massima di un file viene dichiarata al momento della richiesta di creazione dello stesso. In molti casi questo valore può essere stimato in modo affidabile, ad esempio durante una compilazione, la produzione di file di dati riassunto o il trasferimento di un file da un altro sistema attraverso la rete di comunicazione. Però per molte

applicazioni è difficile, se non impossibile, stimare la dimensione massima possibile di un file in modo sicuro. In questi casi, utenti e programmatori di applicazioni tenderebbero a sovrastimare la dimensione del file in modo da non rimanere senza spazio, causando uno spreco, dal punto di vista dell'allocazione della memoria secondaria. Pertanto, ci sono vantaggi nell'uso dell'allocazione dinamica, che alloca spazio ad un file in porzioni, man mano che nuovo spazio diventa necessario.

Dimensione delle porzioni

Il secondo degli aspetti elencati riguarda la dimensione delle porzioni allocate ad un file. Da una parte, potremmo allocare una porzione abbastanza larga da contenere l'intero file, dall'altra, potremmo allocare lo spazio su disco un blocco alla volta. Nello scegliere la dimensione delle porzioni, c'è un compromesso tra l'efficienza dal punto di vista di un singolo file e l'efficienza del sistema nel suo insieme. [WIED87] elenca quattro elementi che devono essere tenuti in considerazione nel cercare questo compromesso:

1. La contiguità di spazio aumenta le prestazioni, specialmente per operazioni Retrieve_Next e per transazioni eseguite in un sistema operativo orientato alle transazioni.
2. Un grande numero di porzioni piccole aumenta la dimensione di tabelle necessarie a gestire le informazioni di allocazione.
3. Porzioni a dimensione fissata (ad esempio, blocchi) semplificano la riallocazione di spazio.
4. Porzioni a dimensione variabile o piccole porzioni a dimensione fissa minimizzano lo spreco di spazio a causa di allocazione in eccesso.

Naturalmente, questi elementi interagiscono e devono essere considerati nel loro insieme. Il risultato è che vi sono due principali alternative:

- **Porzioni grandi variabili e contigue:** questo garantisce prestazioni migliori, in quanto la dimensione variabile evita lo spreco e le tabelle di allocazione di file sono piccole. Però è difficile il riuso di spazio.
- **Blocchi:** porzioni piccole e fissate forniscono una maggiore flessibilità. Esse possono richiedere grandi tabelle o strutture complesse per l'allocazione. Si è abbandonata la contiguità; si allocano i blocchi man mano che servono.

Entrambe le opzioni sono compatibili con preallocazione o allocazione dinamica. Nel primo caso, un gruppo contiguo di blocchi viene preallocato ad un file, eliminando il bisogno di una tabella di allocazione di file; quello che serve è un puntatore al primo blocco e il numero dei blocchi allocati. Nel secondo caso, tutte le porzioni richieste sono allocate in una volta; questo significa che la tabella di allocazione di file rimarrà di dimensione fissata.

Con porzioni di dimensione variabile, dobbiamo preoccuparci della frammentazione dello spazio libero. Questo aspetto è stato affrontato nel caso di partizionamento della memoria centrale nel Capitolo 7. Alcune possibili strategie alternative sono le seguenti:

- **First fit**: scegli il primo gruppo di blocchi non usati contigui di dimensione sufficiente.
- **Best fit**: scegli il gruppo di blocchi non usati di dimensione minore ma sufficiente.
- **Nearest fit**: scegli il gruppo di blocchi non usati di dimensione sufficiente più vicino alla precedente allocazione, per incrementare la località.

Non è chiaro quale sia la strategia migliore. La difficoltà nel modellare strategie alternative consiste nel fatto che vi sono molti fattori che interagiscono, compresi i tipi di file, la struttura degli accessi al file, il grado di multiprogrammazione e altri fattori di prestazioni del sistema, cache di disco, scheduling di disco e così via.

Metodi di allocazione di file

Adesso che abbiamo confrontato preallocazione e allocazione dinamica e analizzato l'aspetto relativo alla dimensione della porzione, siamo nella posizione di considerare specificatamente metodi di allocazione di file; esistono tre metodi di uso comune: contiguo, a catena, indicizzato. La Tabella 12.3 riassume alcune delle caratteristiche di ogni metodo.

Con l'**allocazione contigua**, un unico insieme di blocchi contigui è allocato ad un file al momento della creazione (Figura 12.7); pertanto, questa è una strategia di preallocazione, che usa porzioni di dimensione variabile. La tabella di allocazione di file ha bisogno di un solo ingresso per ogni file, che indichi il blocco di inizio e la lunghezza del file. L'allocazione contigua è la migliore forma di allocazione dal punto di vista di un singolo file sequenziale. Più blocchi alla volta possono essere portati in memoria allo scopo di migliorare le prestazioni dell'I/O per l'elaborazione sequenziale. Inoltre è facile recuperare un blocco singolo; per esempio, se un file inizia al blocco b e si sta cercando l'i-esimo blocco del file, la sua locazione in memoria secondaria sarà semplicemente $b+i$. L'allocazione contigua presenta alcuni problemi, in quanto produce frammentazione esterna e rende difficile il ritrovamento di blocchi liberi contigui di dimensione sufficiente; sarà necessario, ogni tanto, eseguire un algoritmo di compattazione che liberi spazio aggiuntivo su disco (Figura 12.8). Inoltre, con la preallocazione,

Tabella 12.3 *Metodi di allocazione dei file*

	Contiguo	A catena	Indicizzato	
Preallocazione?	Necessaria	Possibile	Possibile	
Dimensione delle porzioni fisse o variabili?	Variabile	Blocchi fissi	Blocchi fissi	Variabile
Dimensione della porzione	Grande	Piccola	Piccola	Media
Frequenza di allocazione	Una volta	Bassa o alta	Alta	Bassa
Tempo di allocazione	Medio	Lungo	Breve	Medio
Dimensione della tabella di allocazione dei file	Una entry	Una entry	Grande	Media

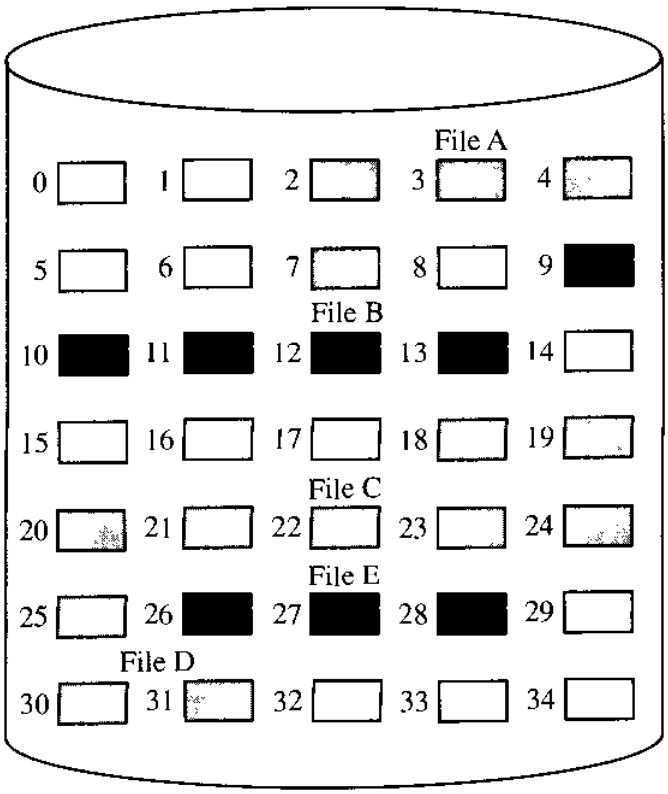


Tabella di allocazione dei file

Nome del file	Blocco di inizio	lunghezza
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figura 12.7 *Allocazione contigua*

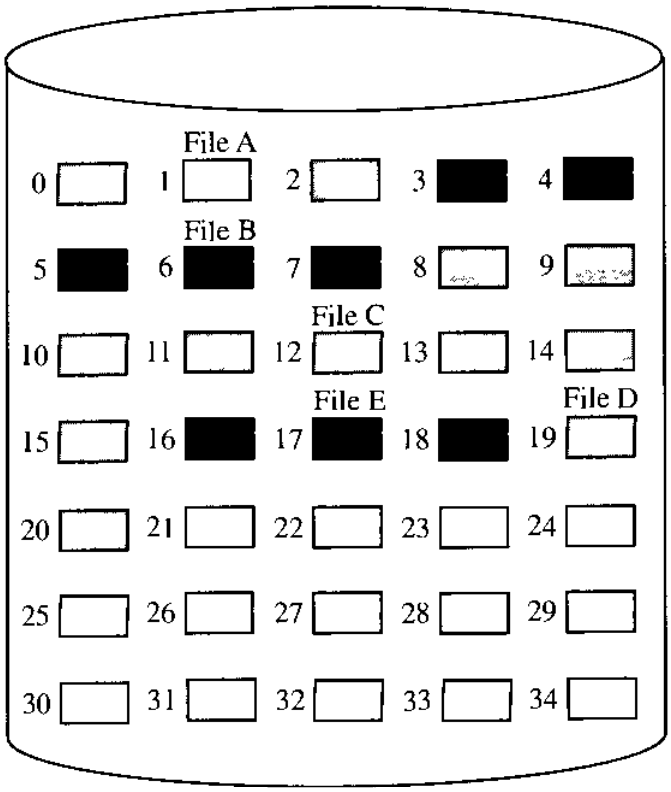


Tabella di allocazione dei file

Nome del file	Blocco di inizio	lunghezza
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figura 12.8 *Allocazione contigua (dopo la compattazione)*

è necessario dichiarare la dimensione del file al momento della creazione, con i problemi menzionati in precedenza.

All'estremo opposto dell'allocazione contigua si trova l'**allocazione a catena** (Figura 12.9). In pratica, l'allocazione avviene sulla base di un blocco singolo. Ogni blocco contiene un puntatore al blocco successivo nella catena. Anche in questo caso la FAT serve solo per fornire l'ingresso iniziale di ciascun file, e conterrà il blocco iniziale e la lunghezza del file. Anche se è possibile effettuare preallocazione, più comunemente si allocheranno i blocchi man mano che questi diventano necessari. Scegliere un blocco è a questo punto una questione molto semplice: ogni blocco libero può venire aggiunto alla catena; non ci si deve preoccupare più del problema della frammentazione esterna, in quanto si ha bisogno di un solo blocco alla volta. Questo tipo di organizzazione fisica si adatta meglio a file sequenziali che vengono elaborati sequenzialmente. Selezionare un singolo blocco di un file, richiede seguire la catena fino al blocco desiderato.

Una conseguenza dell'allocazione a catena, così come è stata descritta finora, è che non lascia spazio al principio di località, pertanto se è necessario portare in memoria diversi blocchi di un file allo stesso tempo, come avviene nell'elaborazione sequenziale, si dovranno effettuare una serie di accessi a differenti parti del disco. Questo è forse un effetto più significativo in un sistema a singolo utente, ma può diventare anche un problema per sistemi condivisi; per risolvere questo effetto, alcuni sistemi periodicamente compattano i file (Figura 12.10).

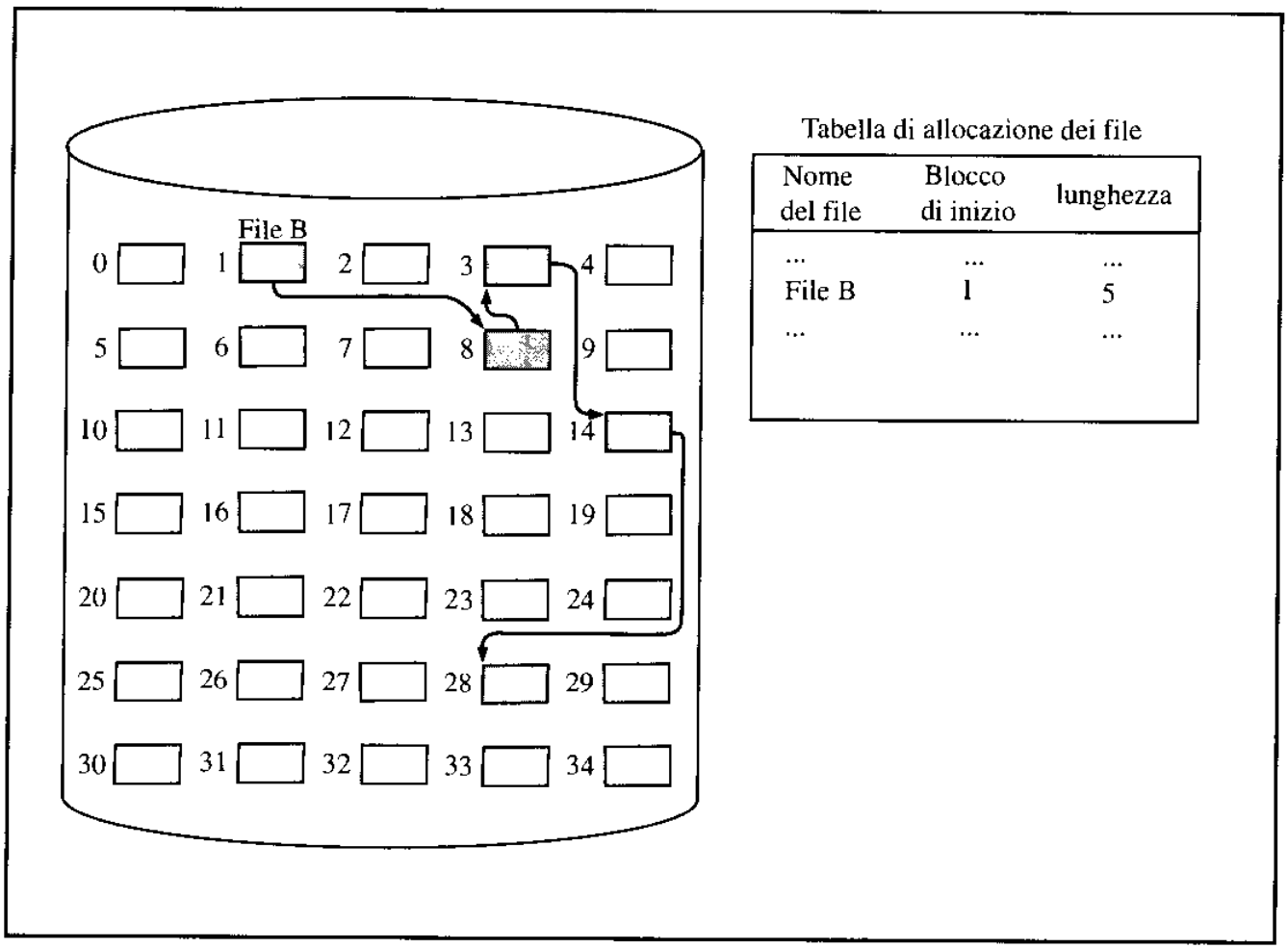


Figura 12.9 *Allocazione a catena*

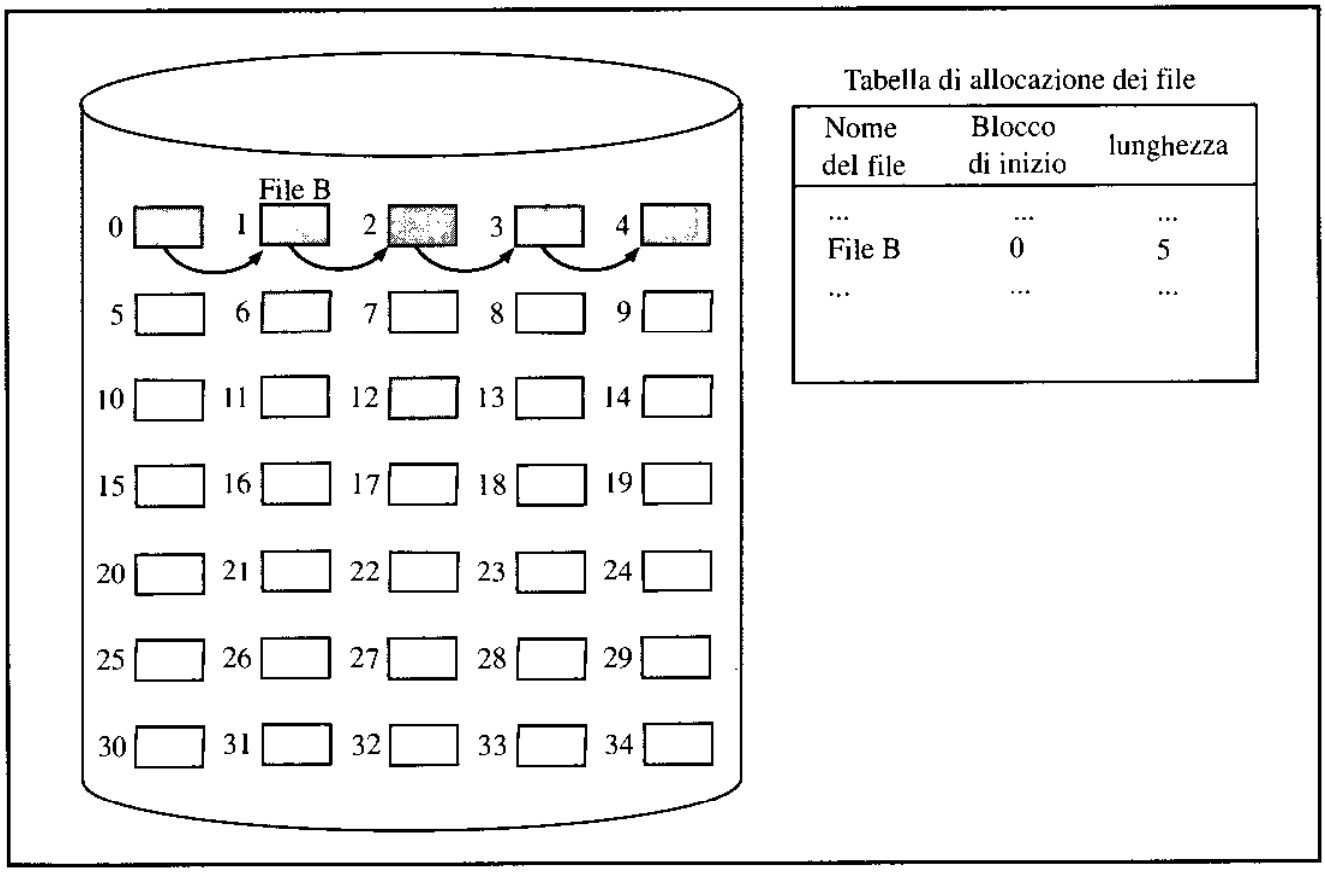


Figura 12.10 *Allocazione a catena (dopo la compattazione)*

L'**allocazione indicizzata** affronta molti dei problemi delle allocazioni contigua e a catena. In questo caso, la FAT contiene un livello di indicizzazione separata per ogni file, contenente un ingresso per ogni porzione allocata al file. Solitamente, i file indice non vengono fisicamente immagazzinati come parte della FAT, ma tenuti su un blocco separato, a cui punta l'entry della FAT relativa al file in questione. L'allocazione può avvenire sia sulla base di blocchi a dimensione fissa (Figura 12.11) che su porzioni a dimensione variabile (Figura 12.12). L'allocazione a blocchi elimina la frammentazione esterna, mente l'allocazione con porzioni a dimensione variabile migliora la località; in entrambi i casi, possono essere necessarie saltuarie compattazioni di file. La compattazione di file, riduce la dimensione dell'indice nel caso di porzioni a dimensione variabile, ma non nel caso di allocazione a blocchi. L'allocazione indicizzata supporta sia l'accesso sequenziale sia l'accesso diretto al file, e pertanto è la forma più popolare di allocazione di file.

Gestione dello spazio libero

Come è necessario gestire lo spazio assegnato ai file, è necessario gestire anche lo spazio che correntemente non è allocato a nessun file. Per eseguire una qualunque delle tecniche di allocazione precedentemente descritte, è necessario sapere quali blocchi del disco sono disponibili, abbiamo quindi bisogno di una **tabella di allocazione del disco** in aggiunta alla tabella di allocazione dei file. Tre tecniche sono di uso comune: la tabella di bit, le porzioni libere a catena e l'indicizzazione.

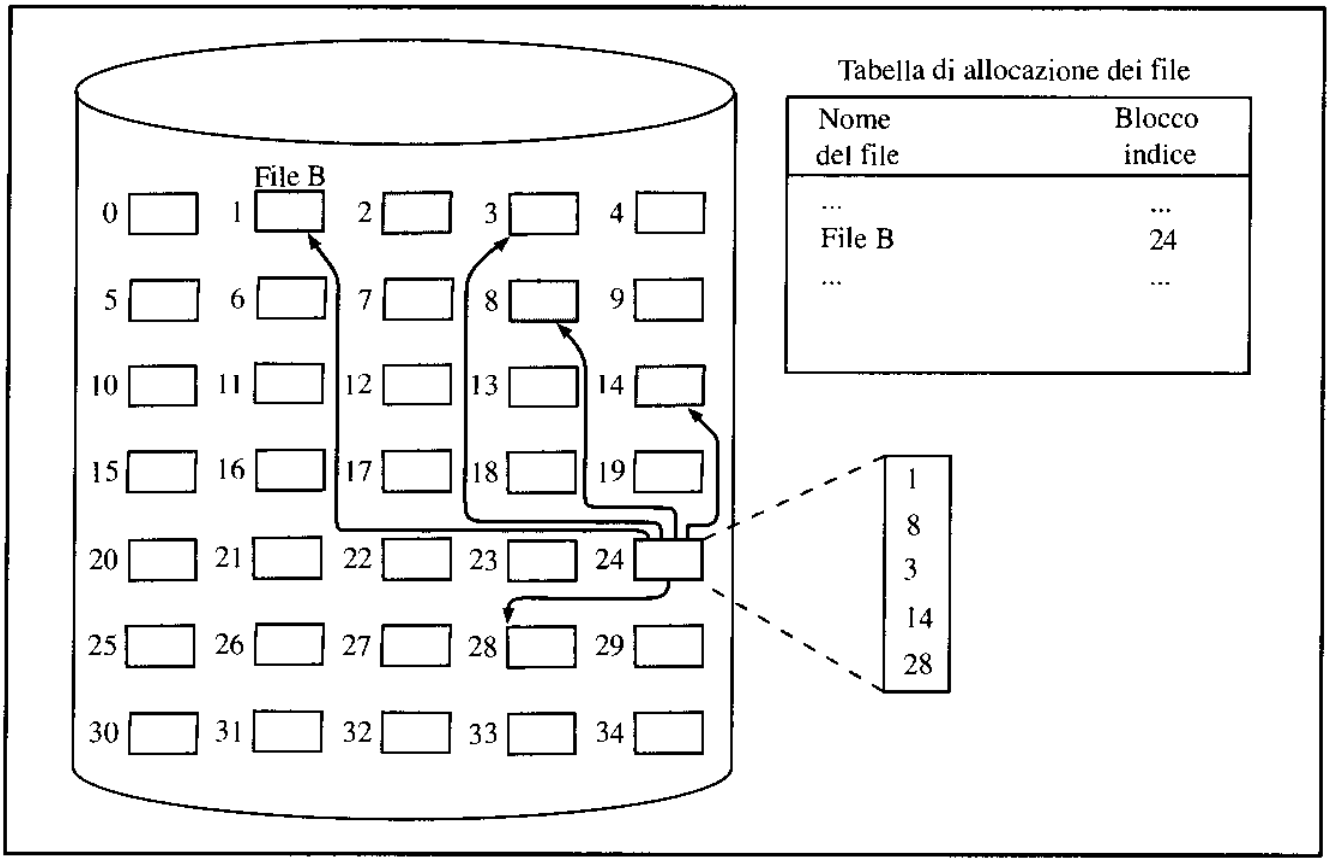


Figura 12.11 *Allocazione indicizzata con porzioni da blocchi*

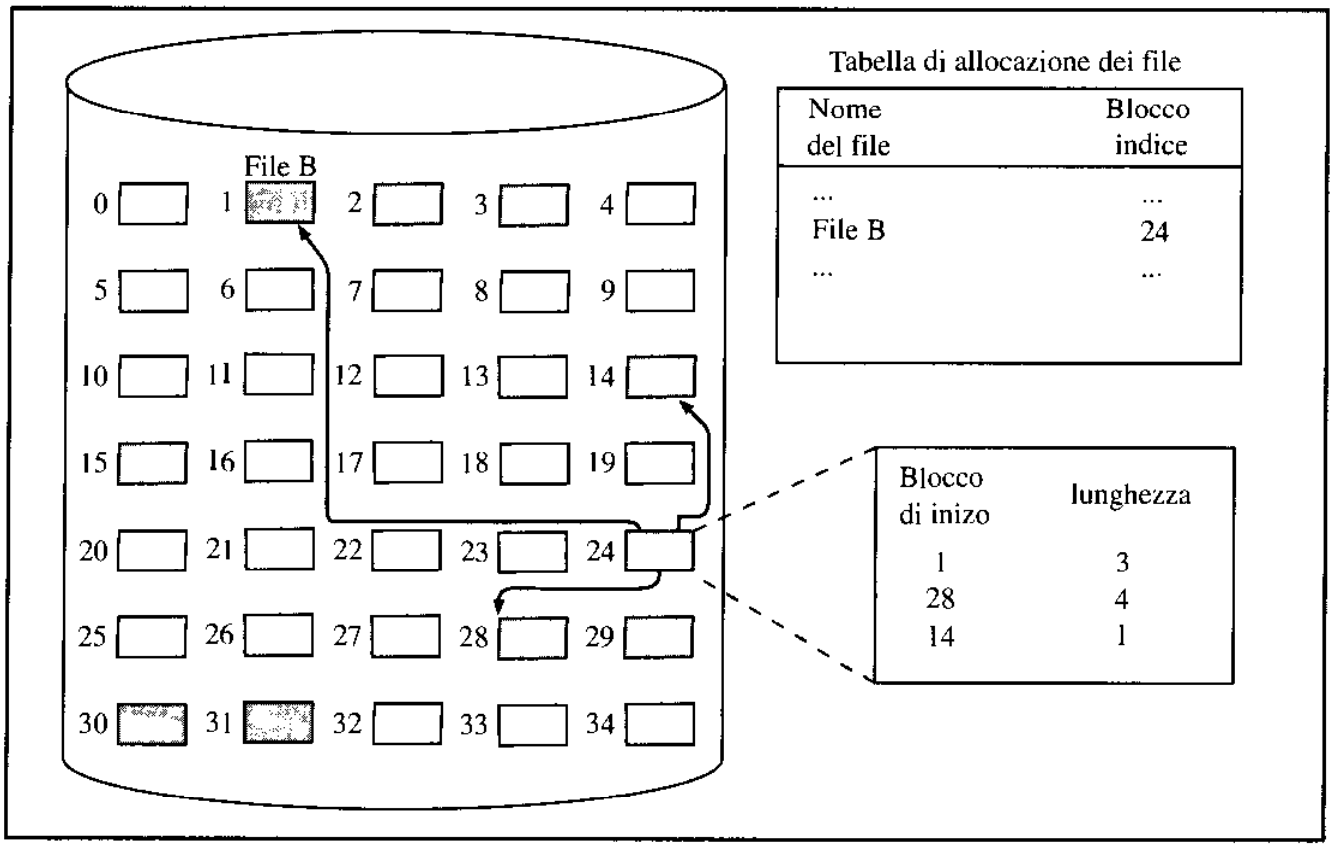


Figura 12.12 *Allocazione indicizzata con porzioni a lunghezza variabile*

Tabelle di bit

Questo metodo usa un vettore contenente un bit per ogni blocco sul disco. Ogni elemento a 0, corrisponde ad un blocco libero e ogni 1 corrisponde ad un blocco in uso. Per esempio, per lo schema di disco della Figura 12.7, occorre un vettore di 35 elementi con il seguente valore:

00111000011111000011111111111011000

Una tabella di bit ha il vantaggio di rendere facile il recupero di uno o un gruppo di blocchi liberi contigui, pertanto le tabelle di bit funzionano bene con ognuno dei metodi di allocazione di file descritti. Un altro vantaggio è dato dal fatto che è la struttura più piccola, e che quindi può essere tenuta in memoria centrale, evitando il bisogno di portare le informazioni di allocazione di disco in memoria ogni volta che si effettua un'allocazione.

Porzioni libere a catena

Le porzioni libere possono venire concatenate insieme usando un puntatore e un valore di lunghezza di ogni porzione libera. Questo metodo richiede un utilizzo di spazio trascurabile in quanto non c'è bisogno di una tabella di allocazione del disco, ma solo dello spazio necessario a tenere il puntatore all'inizio della catena e la lunghezza della prima porzione. Questo metodo si adatta a tutti i metodi di allocazione; se l'allocazione avviene un blocco alla volta, si può semplicemente scegliere il blocco libero all'inizio della catena e aggiornare il primo puntatore o il valore di lunghezza. Se l'allocazione avviene a porzioni di lunghezza variabile, si può usare un algoritmo first-fit: si ricuperano le intestazioni delle porzioni, una alla volta, per determinare la prima porzione libera nella catena dalla dimensione adatta e anche in questo caso si devono aggiornare il puntatore e il valore di lunghezza.

Indicizzazione

L'approccio ad indicizzazione tratta lo spazio libero come un file e usa una tabella di indici come descritto nel caso di allocazione di file. Per ragioni di efficienza, l'indice dovrebbe essere sulla base di porzioni a dimensione variabile piuttosto che blocchi. Pertanto c'è un ingresso nella tabella per ogni porzione libera del disco. Questo approccio fornisce un supporto efficiente per tutti i metodi di allocazione di file.

Affidabilità

Consideriamo il seguente scenario:

1. L'utente A richiede un'allocazione di file da aggiungere ad un file esistente.
2. La richiesta è soddisfatta e le tabelle di allocazione di disco e di file sono aggiornate in memoria centrale, ma non ancora su disco.
3. C'è un crash di sistema, dopodiché il sistema riparte.

4. L'utente B richiede un'allocazione di file e gli viene allocato dello spazio su disco che si sovrappone all'ultima allocazione dell'utente A.
5. L'utente A accede ad una porzione della sovrapposizione tramite un riferimento che si trova all'interno di un file di A.

Questo accade perché il sistema, per efficienza, mantiene una copia della tabella di allocazione del disco e della FAT su disco, e un'altra in memoria centrale. Per evitare questo tipo di errore, dovrebbero essere eseguiti i seguenti passi, nel momento in cui si richiede un'allocazione di file:

1. Bloccare la tabella di allocazione di disco sul disco, impedendo ad un altro utente di alterare la tabella finché questa allocazione non è stata completata.
2. Cercare dello spazio disponibile nella tabella di allocazione di disco, supponendo che una copia della tabella di allocazione di disco venga sempre tenuta in memoria centrale; altrimenti la tabella deve prima essere copiata in memoria.
3. Allocare lo spazio, aggiornare la tabella di allocazione di disco e aggiornare il disco, riscrivendo la tabella di allocazione sul disco; inoltre, in caso di allocazione di disco a catena, occorre aggiornare alcuni puntatori su disco.
4. Aggiornare la tabella di allocazione di file e aggiornare il disco.
5. Sbloccare la tabella di allocazione di disco.

Questa tecnica eviterà errori, ma quando si allocano frequentemente piccole porzioni, l'impatto in termini di prestazioni sarà sostanziale. Per ridurre tale sovraccarico, si potrebbe usare uno schema di allocazione a lotti; in questo caso si ottiene un lotto di porzioni libere sul disco, che sono marcate come "in uso". Al momento dell'allocazione di una porzione di questo lotto, l'allocazione può avvenire in memoria centrale. Quando il lotto è esaurito, la tabella di allocazione di disco è aggiornata sul disco, e un nuovo lotto può essere acquisito. Se avviene un crash di sistema, le porzioni di disco marcate "in uso" devono essere pulite prima di venire riutilizzate; la tecnica di pulizia dipenderà dalle caratteristiche particolari del sistema operativo.

12.7 Gestione del file in UNIX

Il nucleo di UNIX vede tutti i file come flussi di byte. Ogni struttura logica interna è specifica dell'applicazione. Però UNIX si interessa della struttura fisica dei file.

Si distinguono quattro tipi di file:

- **Ordinari:** file che contengono informazioni inserite dagli utenti, da un programma applicativo o da un programma di sistema.
- **Directory:** contengono una lista di nomi di file più puntatori agli inodi (nodi indice) associati, che descriveremo in seguito. Le directory sono organizzate gerarchicamente (Figura 12.4).

I file directory non sono altro che file ordinari con speciali protezioni in scrittura, in modo che solo il file system possa scriverli, mentre l'accesso in lettura è concesso ai programmi utente.

- **Speciali:** usati per accedere a dispositivi periferici, quali terminali e stampanti. Ad ogni dispositivo di I/O è associato un file speciale, come discusso nella Sezione 11.7.
- **File con nome**: pipe con nome come discusso nella Sezione 6.7.

In questa sezione ci occupiamo della manipolazione di file ordinari, che corrispondono a quelli che la maggior parte dei sistemi trattano come file.

Inodi

Tutti i tipi di file UNIX sono amministrati dal sistema operativo tramite inodi. Un inodo (nodo informazione) è una struttura di controllo che contiene le informazioni su un file di cui il sistema operativo ha bisogno. Molti nomi di file possono essere associati allo stesso inodo, ma un inodo attivo è associato ad esattamente un file, e ogni file è controllato da esattamente un inodo.

Gli attributi del file e i suoi permessi, oltre che altre informazioni di controllo, sono immagazzinati nell'inodo; la Tabella 12.4 mostra una lista dei contenuti.

Tabella 12.4 *Informazioni contenute all'interno di un inodo UNIX residente su disco*

Modo del file	Flag di 16 bit che memorizza i permessi di accesso ed esecuzione associati al file.
	12-14 tipo di file (regolare, directory, speciale a caratteri o a blocchi, pipe FIFO)
	9-11 flag di esecuzione
	8 permessi in lettura del proprietario
	7 permessi in scrittura del proprietario
	6 permessi di esecuzione del proprietario
	5 permessi in lettura del gruppo
	4 permessi in scrittura del gruppo
	3 permessi di esecuzione del gruppo
	2 permessi in lettura degli altri utenti
	1 permessi in scrittura degli altri utenti
	0 permessi di esecuzione degli altri utenti
Contatore dei link	Numero di riferimenti a questo inodo nelle directory
ID del proprietario	Identificatore del proprietario del file
ID del gruppo	Identificatore del gruppo a cui appartiene il proprietario del file
Dimensione del file	Numero di byte del file
Indirizzi del file	39 byte di informazioni di indirizzamento
Ultimo accesso	Tempo dell'ultimo accesso
Ultima modifica	Tempo dell'ultima modifica al file
Modifica dell'inodo	Tempo dell'ultima modifica all'inodo

Allocazione di file

L'allocazione di file è basata su blocchi, ed è dinamica in base alle richieste, piuttosto che basata su preallocazione, quindi i blocchi di un file su disco non sono necessariamente contigui. Si utilizza un metodo indicizzato per tenere traccia di ogni file, dove parte dell'indice è immagazzinato nell'inodo del file. L'inodo contiene 39 byte di informazioni di indirizzamento, organizzati in 13 indirizzi, o puntatori, da tre byte ciascuno. I primi 10 indirizzi puntano ai primi 10 blocchi dati del file. Se il file è più lungo di 10 blocchi, allora verranno usati uno o più livelli di indirezione, come spiegato di seguito:

- L'undicesimo indirizzo dell'inodo punta ad un blocco su disco che contiene la successiva porzione dell'indice. Questa è detta blocco singolo indiretto e contiene puntatori ai successivi blocchi del file.
- Se il file contiene più blocchi, il dodicesimo indirizzo dell'inodo punta ad un blocco indiretto doppio. Questo blocco contiene una lista di indirizzi a blocchi indiretti singoli, ognuno dei quali contiene puntatori a blocchi del file.
- Se il file contiene ancora più blocchi, il tredicesimo indirizzo dell'inodo punta ad un blocco indiretto triplo che è un terzo livello di indicizzazione; questo blocco punta a ulteriori blocchi indiretti doppi.

Tutto questo è mostrato in Figura 12.13. Il numero totale di blocchi di dati dipende dalla capacità dei blocchi di dimensione fissa del sistema. In UNIX System V, la lunghezza di un blocco è di 1 Kilobyte, dunque ogni blocco può contenere un totale di 256 indirizzi di blocco, pertanto la dimensione massima di un file, seguendo questo schema, è 16 gigabyte (Tabella 12.5).

Questo schema ha molti vantaggi:

- L'inodo ha dimensione fissata ed è relativamente piccolo e quindi può essere tenuto in memoria centrale per un lungo periodo.
- Si può accedere a file più piccoli con poca o senza indirezione, riducendo il tempo di elaborazione e il tempo di accesso a disco.
- La dimensione massima teorica di un file è sufficientemente grande da soddisfare quasi tutte le applicazioni.

Tabella 12.5 *La capacità di un file UNIX*

Livello	Numero di blocchi	Numero di byte
Diretto	10	10 K
Indiretto singolo	256	256 K
Indiretto doppio	256 × 256 = 65 K	65 M
Indiretto triplo	256 × 65 K = 16 M	16 G

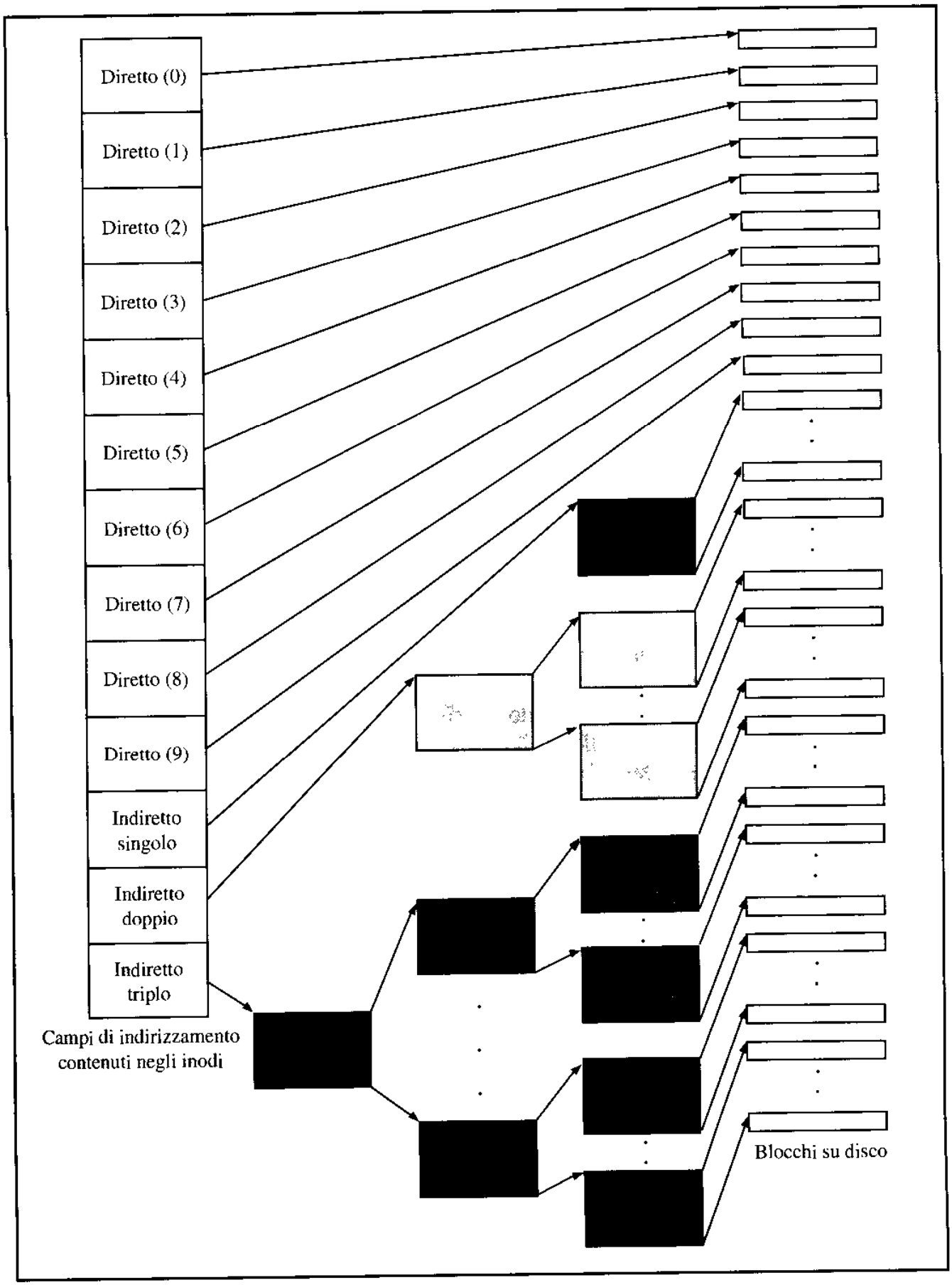


Figura 12.13 *Schema di indirizzamento a blocchi di UNIX*

12.8 Il file system di Windows NT

Windows NT supporta un certo numero di file system, compresa la FAT (*tabella di allocazione di file*, File Allocation Table) di Windows 95, MS-DOS e OS/2, ma i progettisti di Windows NT hanno anche sviluppato un nuovo file system, NTFS (*file system di NT*, NT File System), allo scopo di soddisfare le richieste più consistenti di workstation e server. Esempi di applicazioni di questo tipo sono le seguenti:

- Applicazioni client/server quali server di file, server di calcolo e server di database
- Applicazioni scientifiche e ingegneristiche che richiedono grandi quantità di risorse
- Applicazioni di rete per grandi sistemi aziendali.

Questa sezione fornirà una panoramica di NTFS.

Caratteristiche principali di NTFS

NTFS è un file system flessibile e potente costruito sul modello di file system elegantemente semplice. Le caratteristiche di NTFS maggiormente degne di nota sono le seguenti:

- **Capacità di recupero:** la capacità del file system di recuperare da crash di sistema o guasto di un disco, era prioritaria nella lista dei requisiti per il nuovo file system di Windows NT. Nel caso di questi guasti, NTFS è in grado di ricostruire volumi disco e riportarli in uno stato consistente, utilizzando un modello di elaborazione di transazioni per i cambiamenti del file system; ogni cambiamento significativo viene trattato come un'azione atomica che può venire, o eseguita completamente, o non eseguita affatto. Ogni transazione che era in corso al momento del guasto è conseguentemente annullata o portata a completamento. Inoltre, NTFS usa un meccanismo di memorizzazione ridondante per i dati critici del file system, cosicché il guasto di un settore di disco non causa la perdita di dati che descrivono la struttura e lo stato del file system.
- **Sicurezza:** NTFS usa il modello a oggetti di Windows NT per rafforzare la sicurezza. Un file aperto viene implementato come un oggetto file con un descrittore di sicurezza che ne definisce gli attributi di sicurezza.
- **Dischi grandi e file grandi:** NTFS supporta dischi molto grandi e file molto grandi, più efficientemente di ogni altro file system, compresa la FAT.
- **Flussi multipli di dati:** il contenuto effettivo di un file è trattato come un flusso di byte. In NTFS è possibile definire flussi multipli di dati per un singolo file. Un esempio dell'utilità di questa caratteristica è data dal fatto che permette a Windows NT di venire usato da sistemi Macintosh remoti per immagazzinare e recuperare dati. Su Macintosh ogni file ha due componenti: i dati del file e i dati informativi sul file (*resource fork*). NTFS tratta queste due componenti come due flussi di dati.

- **Capacità di indicizzazione generale:** NTFS associa una collezione di attributi ad ogni file. L'insieme delle descrizioni dei file nel sistema di gestione dei file è organizzato come un database relazionale, in modo tale che i file possano essere indicizzati rispetto ad ognuno degli attributi.

Il volume di NTFS e la struttura dei file

NTFS utilizza i seguenti concetti di memorizzazione su disco:

- **Settore:** la più piccola unità di memorizzazione fisica sul disco. La dimensione dei dati, in byte, è una potenza di 2 ed è quasi sempre 512.
- **Cluster:** uno o più settori contigui (vicini sulla stessa traccia). La dimensione del cluster in settori è una potenza di 2.
- **Volume:** una partizione logica del disco, consistente di uno o più cluster usata dal file system per allocare spazio. In ogni istante, un volume si compone di informazioni del file system, una collezione di file e spazio non allocato che può essere allocato a file. Un volume può essere una porzione o un intero disco singolo o può estendersi attraverso dischi multipli. Se si usa RAID 5, hardware o software, un volume consiste di strisce che si estendono su più dischi. La dimensione massima del volume per NTFS è 2^{64} byte.

Il cluster è l'unità fondamentale di allocazione di NTFS, il quale non riconosce i settori. Per esempio, supponiamo che un settore sia di 512 byte e che il sistema sia configurato con due settori per cluster (un cluster = 1 kilobyte). Se un utente crea un file di 1600 byte, verranno allocati 2 cluster a tale file. Più tardi, se l'utente aggiorna il file a 3200 byte, verranno allocati altri due cluster. I cluster allocati ad un file non devono per forza essere contigui; è possibile frammentare un file sul disco. Al momento, la dimensione massima di file supportata da NTFS è di 2^{32} cluster, che è equivalente ad un massimo di 2^{48} byte.

L'uso di cluster per l'allocazione rende NTFS indipendente dalla dimensione fisica del settore, permettendo a NTFS di supportare facilmente dischi non standard che non hanno una dimen-

Tabella 12.6 *Partizioni di Windows NTFS e dimensioni dei cluster*

Dimensione del volume	Settori per cluster	Dimensione del cluster
≤ 512 Mbyte	1	512 byte
512 Mbyte - 1 Gbyte	2	1 K
1 Gbyte - 2 Gbyte	4	2 K
2 Gbyte - 4 Gbyte	8	4 K
4 Gbyte - 8 Gbyte	16	8 K
8 Gbyte - 16 Gbyte	32	16 K
16 Gbyte - 32 Gbyte	64	32 K
> 32 Gbyte	128	64 K

sione del settore di 512 byte, e di supportare efficientemente dischi molto grandi e file molto grandi, utilizzando una dimensione maggiore per i cluster. L'efficienza deriva dal fatto che il file system deve tenere traccia di ogni cluster allocato ad ogni file; con cluster più grandi, c'è un numero minore di elementi da gestire.

La Tabella 12.6 mostra le dimensioni per difetto dei cluster di NTFS, che dipendono dalla dimensione del volume. La dimensione del cluster usata per un particolare volume è stabilita da NTFS quando l'utente chiede di formattare un volume.

Struttura del volume di NTFS

NTFS usa un approccio semplice ma potente all'organizzazione delle informazioni su un volume di disco. Ogni elemento di volume è un file, ed ogni file consiste in una collezione di attributi, persino i dati contenuti nel file sono trattati come un attributo. Con questa semplice struttura, poche funzioni generali sono sufficienti ad organizzare e gestire un file system.

La Figura 12.14 mostra la struttura di un volume di NTFS, che consiste di quattro regioni. I primi settori del volume sono occupati dal settore delle **partizioni di avvio** (in inglese, partition boot sector; anche se è chiamato settore, può occupare fino a 16 settori), che contiene informazioni sulla struttura del volume e le strutture del file system, oltre che informazioni e codici per l'avvio. Questo è seguito dalla **MFT** (*tabella di file master*, Master File Table), che contiene informazioni su tutti i file e le cartelle (directory) di questo volume di NTFS oltre che informazioni sullo spazio disponibile non allocato. In sostanza, la MFT è una lista di tutti i contenuti di questo volume NTFS, organizzata come un insieme di righe in una struttura a database relazionale.

Dopo la MFT, vi è una regione, solitamente lunga 1 megabyte, contenente i **file di sistema**; tra i file di questa regione, troviamo i seguenti:

- **MFT2:** una replica delle prime tre righe della MFT, usate per garantire l'accesso alla MFT in caso di guasto di un singolo settore.
- **Log file:** una lista di passi di transazione usata da NTFS nella sua funzionalità di recupero.
- **Bit map dei cluster:** una rappresentazione del volume, che mostra quali sono i cluster in uso.
- **Tabella di definizione degli attributi:** definisce i tipi di attributo supportati su questo volume, e indica se possono essere indicizzati e se possono essere recuperati durante un'operazione di recupero di sistema.

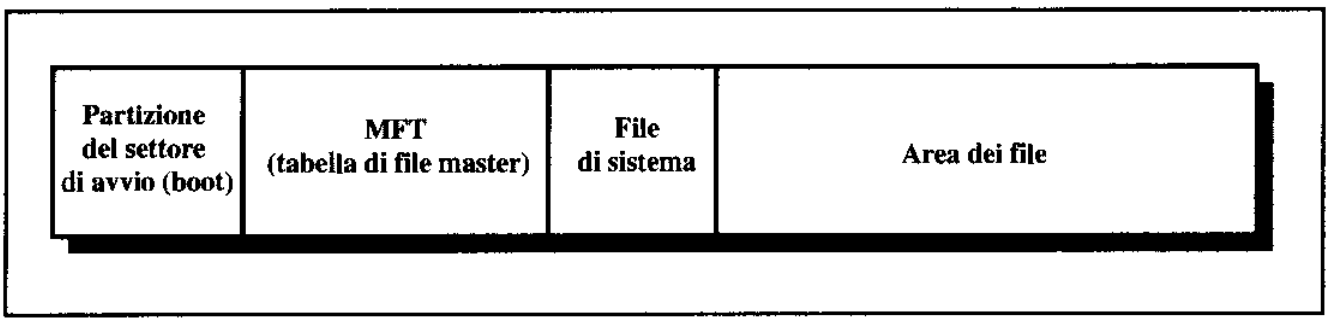


Figura 12.14 *Struttura del volume di NTFS*

Tabella di file master MFT

Il cuore del file system di Windows NT è la MFT che è organizzata come una tabella con righe a lunghezza variabile, chiamate record. Ogni riga descrive un file o una cartella del volume in questione, compresa la MFT stessa, che è trattata come un file. Se i contenuti di un file sono sufficientemente piccoli, allora l'intero file si trova in una riga della MFT, altrimenti, la riga relativa a questo file contiene informazioni parziali e la parte rimanente del file si distribuisce in altri cluster disponibili del volume, con puntatori a questi cluster mantenuti nella riga della MFT relativa al file.

Ogni record della MFT si compone di un insieme di attributi che servono a definire le caratteristiche del file (o della cartella) e i contenuti del file. La Tabella 12.7 contiene una lista degli attributi che possono venire trovati in una riga, dove gli attributi ombreggiati sono obbligatori.

Capacità di recupero

Il NTFS rende possibile il recupero del file system ad uno stato consistente in seguito a un crash di sistema o a un guasto di un disco. Gli elementi chiave che supportano questa capacità, sono i seguenti (Figura 12.15):

- **Gestore dell'I/O:** contiene il driver di NTFS, che si occupa delle funzioni base di apertura, chiusura, lettura e scrittura di NTFS. In aggiunta ad esso, può essere configurato per l'uso il modulo FTDISK del RAID software.

Tabella 12.7 *Tipi degli attributi di file e directory in Windows NTFS*

Tipo degli attributi	Descrizione
Informazione standard	**comprende gli attributi di accesso (sola lettura, lettura/scrittura, ecc.); timestamp, tra cui quando il file è stato creato e l'ultima volta che è stato modificato; quante directory puntano al file (contatore dei link).**
Lista degli attributi	Una lista degli attributi che caratterizzano il file e il suo riferimento all'interno della MFT, dove tali attributi sono localizzati. Usato quando non tutti gli attributi entrano nel singolo record della MFT.
Nome del file	**un file o una directory devono avere uno o più nomi.**
Descrittore sicurezza	**specifica il proprietario del file e quali altri utenti possono accedervi.**
Dati	**Il contenuto del file; un file ha un attributo dati per difetto senza nome, e può avere uno o più attributi dati con nome.**
Radice dell'indice	Usato per implementare le cartelle.
Allocazione dell'indice	Usato per implementare le cartelle.
Informazioni sul volume	Include informazioni relative al volume, quali la versione e il nome del volume.
Bitmap	Fornisce una mappa che rappresenta i record in uso nella MFT o nella cartella

(***Nota**: le righe ombreggiate si riferiscono ad attributi obbligatori, le altre ad attributi facoltativi.*)

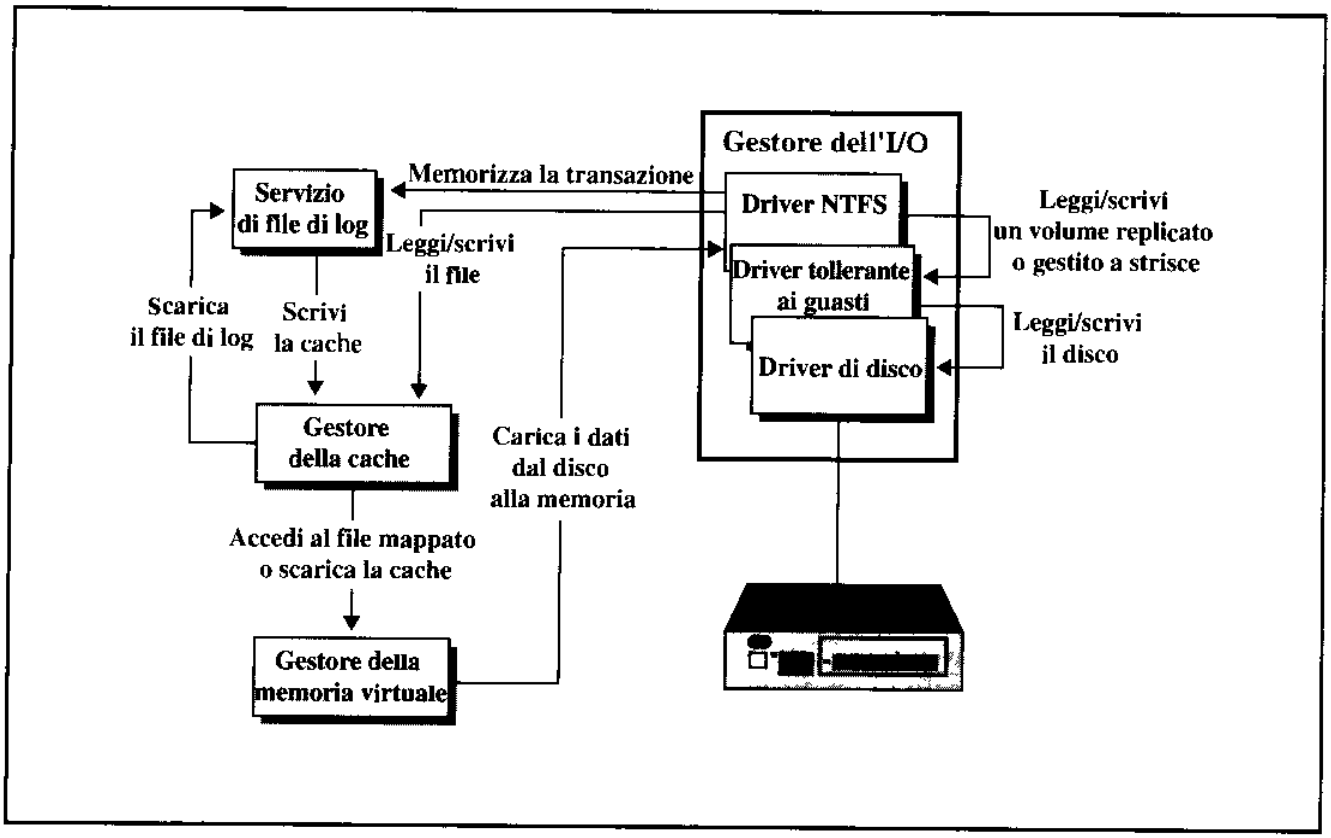


Figura 12.15 *Componenti di Windows NTFS [CUST94]*

- **Servizio di log file:** mantiene un log delle scritture su disco. Il log file viene usato per recuperare un volume formattato NTFS, nel caso di guasto di disco.
- **Gestore della cache:** è responsabile di gestire la cache per letture e scritture di file, allo scopo di migliorare le prestazioni. Il gestore della cache ottimizza l'I/O di disco usando le tecniche di scrittura pigra e cessione pigra, come descritto nella Sezione 11.8.
- **Gestore della memoria virtuale:** NTFS accede a file nella cache mappando i riferimenti ai file in riferimenti alla memoria virtuale e leggendo e scrivendo la memoria virtuale.

È importante notare che le procedure di recupero usate da NTFS sono progettate per recuperare i dati relativi al file system, non il contenuto dei file, pertanto, l'utente non dovrebbe perdere un volume o la struttura directory/file di un'applicazione a causa di un crash. Comunque i dati dell'utente non vengono garantiti dal file system; il garantire un pieno recupero, compreso il recupero dei dati utente, renderebbe la funzionalità di recupero troppo elaborata e la porterebbe ad utilizzare troppe risorse.

L'essenza della capacità di recupero di NTFS è l'uso dei file di log. Ogni operazione che altera il file system è trattata come una transazione. Ogni sottooperazione di una transazione che alteri importanti strutture dati del file system è registrata in un log file prima di essere registrata nel volume del disco. Usando il log, una transazione parzialmente completata al momento di un crash può in seguito venire terminata o annullata, quando il sistema è recuperato.

In termini generali, vengono eseguiti i seguenti passi per garantire il recupero, come descritto in [CUST94]:

- NTFS prima di tutto chiama il log file system per registrare nel log file della cache ogni transazione che modificherà la struttura del volume.
- NTFS modifica il volume (nella cache).
- Il gestore della cache chiama il log file system chiedendogli di mettere il log file su disco.
- Nel momento in cui gli aggiornamenti del log file sono sicuri sul disco, il gestore di cache fa le modifiche al volume su disco.

12.9 Sommario

Un sistema di gestione dei file è un software di sistema che fornisce servizi ad utenti e applicazioni nell'uso di file, compreso l'accesso a file, la memorizzazione delle directory e il controllo degli accessi. Il sistema di gestione dei file viene solitamente visto come un servizio di sistema che viene a sua volta servito dal sistema operativo, piuttosto che essere una parte del sistema operativo. Comunque, in ogni sistema, almeno una parte delle funzioni di gestione dei file, vengono eseguite dal sistema operativo.

Un file consiste di una collezione di record. Il modo in cui si può accedere a questi record determina l'organizzazione logica del file e, in un certo senso, la sua organizzazione fisica sul disco. Se un file viene principalmente elaborato nel suo insieme, allora un'organizzazione sequenziale è la più semplice e la più appropriata. Se serve un accesso sequenziale, ma si desidera anche un accesso casuale a singoli file, allora un file sequenziale indicizzato può dare le migliori prestazioni. Se l'accesso al file è principalmente di tipo casuale, allora un file indicizzato o un file hash possono essere le organizzazioni più appropriate.

Qualunque sia la struttura di file scelta, occorre anche un servizio di directory. Questo permette ai file di essere organizzati in modo gerarchico. Questa organizzazione è utile all'utente per tenere traccia dei propri file, ed al sistema di gestione del file per fornire un controllo agli accessi e altri servizi forniti all'utente.

I record di file, anche se di dimensione fissata, generalmente non si adattano alla dimensione dei blocchi fisici di disco. Di conseguenza, un qualche tipo di strategia che organizzi i record in blocchi deve essere usata: si deve determinare un compromesso tra complessità, prestazioni ed utilizzo dello spazio.

Una funzione chiave di ogni schema di gestione di file è la gestione dello spazio su disco; fa parte di questa funzione la strategia per allocare blocchi di disco ad un file. Una varietà di metodi sono stati impiegati, e una varietà di strutture dati sono state usate per tenere traccia dell'allocazione di ogni file. Inoltre, lo spazio su disco non ancora allocato deve essere gestito. Quest'ultima funzione consiste soprattutto nel mantenere della tabelle di allocazione del disco che indichino quali sono i blocchi liberi.

12.10 Letture raccomandate

Esistono molti buoni libri sulla gestione di file; i seguenti focalizzano su sistemi di gestione di file, ma fanno anche riferimento ad aspetti relativi ai sistemi operativi. Forse il più utile è [WIED87], che ha un approccio quantitativo alla gestione dei file e tratta tutti gli aspetti sollevati nella Figura 12.2, dalla schedulazione del disco alla struttura del file. [LIVA90] enfatizza le strutture dei file, fornendo una rassegna completa e piuttosto lunga con analisi comparative di prestazioni. [GROS86] fornisce una descrizione equa di aspetti relativi sia all'I/O di file sia a metodi di accesso a file. Contiene inoltre descrizioni di tipo generale delle strutture di controllo necessaria ad un file system, che forniscono un utile riferimento per valutare la progettazione di un file system. [FOLK92] enfatizza l'elaborazione di file, affrontando aspetti quali il mantenimento, la ricerca e l'ordinamento, la condivisione.

[CUST94] fornisce una buona panoramica dei dettagli del file system di NT. [NAGA97] affronta gli argomenti in maggiore dettaglio.

CURT94 Custer, H. *Inside the Windows NT File System.* Redmond, WA: Microsoft Press, 1994.

FOLK92 Folk, M., e Zoellick, B. *File Structures: A Conceptual Toolkit.* Reading, MA: Addison-Wesley, 1992.

GROS86 Grosshans, D. *File Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice Hall, 1986.

LIVE90 Livadas, P. *File Structures: Theory and Practice.* Englewood Cliffs, NJ: Prentice Hall, 1990.

NAGA97 Nagar, R. *Windows NT File System Internals.* Sebastopol, CA: O. Reilly, 1997.

WIED87 Wiederhold, G. *File Organization for Database Design.* New York: McGraw-Hill, 1987.

12.11 Problemi

12.1 Definire i seguenti:

B = dimensione del blocco

R = dimensione del record

P = dimensione del puntatore al blocco

F = fattore di blocco: numero di record attesi all'interno di un blocco

Dare una formula per F nel caso dei tre metodi descritti in Figura 12.6.

12.2 Uno schema per evitare lo spreco o mancanza di contiguità nell'utilizzare la preallocazione è di allocare porzioni di dimensioni crescenti, man mano che la dimensione del file aumenta, per esempio, iniziando con porzioni di un blocco e raddoppiando la dimensione ad ogni allocazione. Consideriamo un file di n record con un fattore F di blocco e supponiamo che la tabella di allocazione dei file, abbia solo un livello di indicizzazione.

a. Dare un limite superiore al numero di entry della tabella di allocazione dei file, come funzione di F e n.

b. Quale è l'ammontare massimo di spazio allocato ai file che rimane inutilizzato ogni volta?

12.3 Quale organizzazione di file sceglieresteste per massimizzare l'efficienza rispetto alla velocità di accesso, l'uso di spazio di memorizzazione, la facilità di aggiornamento (aggiungere/cancellare/modificare), quando i dati sono:

a. Aggiornati raramente, e frequentemente si accede ad essi in modo casuale?

b. Aggiornati frequentemente, e si accede spesso ad ognuno di essi?

c. Aggiornati frequentemente, e si accede ad essi frequentemente in modo casuale?

12.4 Le directory possono essere implementate o come file speciali l'accesso ai quali è limitato, o attraverso file ordinari. Quali sono i vantaggi e gli svantaggi di entrambi gli approcci?

12.5 Alcuni sistemi operativi possiedono file system strutturati ad albero, ma limitano la profondità della struttura ad albero ad un certo numero di livelli. Che effetto ha questa limitazione sugli utenti? Questa scelta semplifica la progettazione del file system? In che senso?

12.6 Considerare un file system gerarchico in cui lo spazio libero di disco è mantenuto in una lista di spazio libero.

a. Supponiamo che il puntatore alla lista di spazio libero vada perso. Come può il sistema ricostruire la lista?

b. Suggerire uno schema che assicuri che il puntatore non vada mai perso, in seguito ad un singolo guasto di memoria.

12.7 Considerare l'organizzazione di un file UNIX, basata su inodi (Figura 12.13); assumiamo che, in ogni inodo, ci siano 12 puntatori diretti ai blocchi e inoltre puntatori indiretti singoli, doppi e tripli. Inoltre, supponiamo che la dimensione dei blocchi di sistema e quella dei settori di disco siano di 8K. Se il puntatore ai blocchi di disco è di 32 bit, con 8 bit che identificano il disco fisico e 24 bit che identificano il blocco fisico, allora

a. Qual è la dimensione di file massima supportata da questo sistema?

b. Qual è la partizione di file system massima supportata da questo sistema?

c. Supponendo che nessun'altra informazione oltre all'inodo sia in memoria centrale, quanti accessi di disco occorrono per accedere al byte in posizione 13 423 956?