

I MODERNI SISTEMI OPERATIVI

SOMMARIO

PREFAZIONE	XV
PREFAZIONE ALL'EDIZIONE ITALIANA	XIX
1. INTRODUZIONE	1
1.1 CHE COS'È UN SISTEMA OPERATIVO?	3
1.1.1 Il sistema operativo come macchina estesa	3
1.1.2 Il sistema operativo come gestore delle risorse	4
1.2 STORIA DEI SISTEMI OPERATIVI.....	5
1.2.1 La prima generazione (1945-1955): valvole e schede a spinotti	6
1.2.2 La seconda generazione (1955-1965): transistor e sistemi batch	6
1.2.3 La terza generazione (1965-1980): circuiti integrati e multiprogrammazione	8
1.2.4 La quarta generazione (dal 1980 ad oggi): i Personal Computer.....	12
1.2.5 L'ontogenesi riassume la filogenesi	15
1.3 LO ZOO DEI SISTEMI OPERATIVI.....	16
1.3.1 Sistemi operativi per i mainframe.....	16
1.3.2 Sistemi operativi per i server	17
1.3.3 Sistemi operativi multiprocessore	17
1.3.4 Sistemi operativi per i personal computer.....	17
1.3.5 Sistemi operativi real-time	17
1.3.6 Sistemi operativi embedded	18
1.3.7 Sistemi operativi per le smart card	18
1.4 RASSEGNA DI HARDWARE PER CALCOLATORI	18
1.4.1 Processori	19

1.4.2 Memoria	21
1.4.3 I dispositivi di ingresso/uscita.....	25
1.4.4 I bus.....	28
1.5 CONCETTI DI BASE SUI SISTEMI OPERATIVI.....	31
1.5.1 I processi	31
1.5.2 Il deadlock.....	33
1.5.3 La gestione della memoria	34
1.5.4 L'ingresso/uscita.....	34
1.5.5 I file	35
1.5.6 Sicurezza	37
1.5.7 La shell	38
1.5.8 Riciclaggio di concetti	39
1.6 LE CHIAMATE DI SISTEMA	40
1.6.1 Chiamate di sistema per la gestione di processi.....	43
1.6.2 Chiamate di sistema per la gestione dei file.....	45
1.6.3 Chiamate di sistema per la gestione delle directory	46
1.6.4 Miscellanea di chiamate di sistema	48
1.6.5 Le Win32 API di Windows	48
1.7 STRUTTURA DI UN SISTEMA OPERATIVO.....	51
1.7.1 Sistemi monolitici	51
1.7.2 Sistemi a livelli	52
1.7.3 Macchine virtuali	53
1.7.4 Exokernel	55
1.7.5 Il modello client-server	56
1.8 RICERCHE SUI SISTEMI OPERATIVI.....	57
1.9 PANORAMICA SUL RESTO DEL LIBRO.....	59
1.10 UNITÀ DI MISURA DECIMALI.....	59
1.11 SOMMARIO	60
2 PROCESSI E THREAD	65
2.1 INTRODUZIONE AI PROCESSI.....	65
2.1.1 Il modello a processi.....	66
2.1.2 La creazione dei processi.....	67
2.1.3 La terminazione dei processi	69
2.1.4 Gerarchie di processi.....	70
2.1.5 Gli stati dei processi	70
2.1.6 Implementazione dei processi	72
2.2 THREAD	74
2.2.1 Il modello a thread	74
2.2.2 Uso dei thread	77
2.2.3 Implementazione dei thread nello spazio utente	82
2.2.4 Implementazione dei thread nel kernel	84
2.2.5 Implementazione ibrida.....	85
2.2.6 Attivazioni dello schedulatore.....	86

2.2.7 Thread pop-up	87
2.2.8 Trasformare codice a thread singolo in multithread	88
2.3 COMUNICAZIONE FRA PROCESSI.....	91
2.3.1 Corse critiche	91
2.3.2 Sezioni critiche	93
2.3.3 Mutua esclusione con attesa attiva	94
2.3.4 Sospensione e risveglio	98
2.3.5 I semafori	100
2.3.6 I mutex	102
2.3.7 I monitor	104
2.3.8 Lo scambio di messaggi	109
2.3.9 Barriere	112
2.4 PROBLEMI CLASSICI DI COMUNICAZIONE FRA PROCESSI.....	113
2.4.1 Il problema dei filosofi a cena	113
2.4.2 Il problema dei lettori e scrittori	116
2.4.3 Il problema del barbiere che dorme	118
2.5 SCHEDULAZIONE DEI PROCESSI.....	120
2.5.1 Introduzione alla schedulazione	120
2.5.2 Schedulazione nei sistemi batch	126
2.5.3 Schedulazione nei sistemi interattivi	129
2.5.4 Schedulazione nei sistemi real time	135
2.5.5 Politiche e meccanismi	136
2.5.6 Schedulazione dei thread	136
2.6 RICERCA SU PROCESSI E THREAD.....	138
2.7 SOMMARIO	139
 3 DEADLOCK	145
3.1 RISORSE	146
3.1.1 Risorse prerilasciabili e non prerilasciabili	146
3.1.2 Acquisizione di risorse	147
3.2 INTRODUZIONE AI DEADLOCK	149
3.2.1 Condizioni per il deadlock	149
3.2.2 Modelli per le situazioni di stallo	150
3.3 L'ALGORITMO DELLO STRUZZO	152
3.4 IDENTIFICAZIONE E RISOLUZIONE DEI DEADLOCK	153
3.4.1 Identificazione del deadlock con una risorsa per ogni classe	154
3.4.2 Identificazione del deadlock con risorse multiple per ogni classe	155
3.4.3 La risoluzione del deadlock	158
3.5 EVITARE I DEADLOCK	159
3.5.1 Traiettorie di risorsa	159
3.5.2 Stati sicuri e non sicuri	161
3.5.3 L'algoritmo del banchiere per una singola risorsa	162
3.5.4 L'algoritmo del banchiere per risorse multiple	163

3.6	PREVENZIONE DA DEADLOCK	164
3.6.1	Negazione della condizione di mutua esclusione	165
3.6.2	Negazione della condizione hold-and-wait	165
3.6.3	Negazione della condizione di mancanza di prelascio.....	166
3.6.4	Negazione della condizione di attesa circolare.....	166
3.7	ALTRI ARGOMENTI	167
3.7.1	Il blocco a due fasi	167
3.7.2	Stallo senza risorse	168
3.7.3	Starvation	168
3.8	RICERCA SUI DEADLOCK	169
3.9	SOMMARIO	169

4 GESTIONE DELLA MEMORIA

4.1	SISTEMI DI BASE PER LA GESTIONE DELLA MEMORIA	174
4.1.1	Monoprogrammazione senza swapping o paginazione	174
4.1.2	Multiprogrammazione con partizioni fisse	175
4.1.3	Modello della multiprogrammazione	176
4.1.4	Analisi delle prestazioni dei sistemi a multiprogrammazione.....	178
4.1.5	Rilocazione e protezione.....	178
4.2	SWAPPING	180
4.2.1	Gestione della memoria con mappe di bit	182
4.2.2	Gestione della memoria con liste concatenate	183
4.3	MEMORIA VIRTUALE	185
4.3.1	La paginazione	186
4.3.2	Tabelle delle pagine	189
4.3.3	TLB: Translation Lookaside Buffer	193
4.3.4	Tabelle delle pagine invertite	196
4.4	ALGORITMI DI RIMPIAZZAMENTO DELLE PAGINE	197
4.4.1	L'algoritmo ottimale di rimpiazzamento delle pagine	198
4.4.2	L'algoritmo di rimpiazzamento delle pagine "non usate di recente"	198
4.4.3	L'algoritmo di rimpiazzamento delle pagine FIFO	199
4.4.4	L'algoritmo di rimpiazzamento delle pagine "della seconda opportunità"	200
4.4.5	L'algoritmo di rimpiazzamento delle pagine "dell'orologio"	201
4.4.6	L'algoritmo di rimpiazzamento delle pagine LRU	201
4.4.7	La simulazione della LRU via software	203
4.4.8	Il modello Working Set	204
4.4.9	L'algoritmo di rimpiazzamento delle pagine WSClock	208
4.4.10	Riepilogo degli algoritmi di rimpiazzamento delle pagine	209
4.5	MODELLAZIONE DEGLI ALGORITMI DI PAGINAZIONE	211
4.5.1	L'anomalia di Belady	211
4.5.2	Gli algoritmi a pila	211
4.5.3	La stringa delle distanze	214
4.5.4	La predizione dei tassi di fault di pagina	215

4.6 PROBLEMATICHE DI PROGETTO DEI SISTEMI CON PAGINAZIONE	216
4.6.1 Confronto delle politiche di allocazione globali e locali	216
4.6.2 Controllo del carico	218
4.6.3 Dimensione della pagina	219
4.6.4 Spazi separati per istruzioni e dati	220
4.6.5 Pagine condivise	221
4.6.6 Politiche di pulizia	223
4.6.7 Interfaccia di memoria virtuale	223
4.7 QUESTIONI IMPLEMENTATIVI.....	224
4.7.1 Implicazioni della paginazione sul sistema operativo.....	224
4.7.2 Trattamento dei fault di pagina	225
4.7.3 Backup delle istruzioni	226
4.7.4 Blocco delle pagine in memoria	227
4.7.5 Memoria secondaria	227
4.7.6 Separazione tra politica e meccanismo	229
4.8 SEGMENTAZIONE	230
4.8.1 Implementazione della segmentazione pura	234
4.8.2 Segmentazione con paginazione: MULTICS	235
4.8.3 Segmentazione con paginazione: Pentium Intel	238
4.9 RICERCA SULLA GESTIONE DELLA MEMORIA	243
4.10 SOMMARIO	243

5. INPUT/OUTPUT 251

5.1 PRINCIPI DELL'HARDWARE DI I/O	251
5.1.1 Dispositivi di I/O	252
5.1.2 I controllori dei dispositivi	253
5.1.3 I/O mappato in memoria	254
5.1.4 Accesso diretto in memoria	257
5.1.5 Le interruzioni rivisitate	260
5.2 PRINCIPI DEL SOFTWARE DI I/O	264
5.2.1 Scopi del software di I/O	264
5.2.2 I/O programmato	265
5.2.3 I/O guidato dalle interruzioni	267
5.2.4 I/O tramite DMA	267
5.3 I LIVELLI SOFTWARE DI I/O	268
5.3.1 Driver delle interruzioni	268
5.3.2 Driver dei dispositivi	270
5.3.3 Software di I/O indipendente dai dispositivi	273
5.3.4 Software di I/O del livello utente	278
5.4 I DISCHI	279
5.4.1 L'hardware dei dischi	279
5.4.2 La formattazione di un disco	293
5.4.3 Gli algoritmi di schedulazione del braccio del disco	296
5.4.4 Trattamento degli errori	300
5.4.5 Memorizzazione stabile	302

5.5	I CLOCK.....	305
5.5.1	L'hardware dei clock	305
5.5.2	Il software dei clock.....	306
5.5.3	Timer soft	309
5.6	I TERMINALI ORIENTATI A CARATTERE	310
5.6.1	L'hardware dei terminali RS-232	311
5.6.2	Il software di input	313
5.6.3	Il software di output.....	317
5.7	INTERFACCE GRAFICHE UTENTE.....	318
5.7.1	Tastiera, mouse e hardware di display dei personal computer	319
5.7.2	Il software di input.....	322
5.7.3	Il software di output in Windows	323
5.8	TERMINALI DI RETE.....	331
5.8.1	Il sistema X Window	331
5.8.2	Il terminale di rete SLIM.....	334
5.9	GESTIONE DEL CONSUMO DI ENERGIA.....	337
5.9.1	Questioni riguardanti l'hardware	338
5.9.2	Questioni riguardanti i sistemi operativi.....	339
5.9.3	Operazioni degradate	344
5.10	RICERCA NEL CAMPO DELL'I/O	345
5.11	SOMMARIO	346

6. I FILE SYSTEM.....

353

6.1	I FILE	354
6.1.1	Denominazione dei file	354
6.1.2	Struttura dei file	356
6.1.3	Tipi di file.....	357
6.1.4	Accesso ai file	359
6.1.5	Attributi dei file	359
6.1.6	Operazioni sui file	360
6.1.7	Un esempio di programma che usa chiamate di sistema sui file.....	361
6.1.8	File mappati in memoria	364
6.2	LE DIRECTORY	365
6.2.1	Sistemi di directory a singolo livello	365
6.2.2	Sistemi di directory a due livelli	366
6.2.3	Sistemi di directory gerarchici	367
6.2.4	I path name	368
6.2.5	Operazioni sulle directory	370
6.3	IMPLEMENTAZIONE DEL FILE SYSTEM	371
6.3.1	La struttura dei file system	371
6.3.2	Implementazione dei file	372
6.3.3	Implementazione delle directory	376
6.3.4	File condivisi.....	379

6.3.5	Gestione dello spazio su disco.....	381
6.3.6	Affidabilità del file system	386
6.3.7	Prestazioni del file system.....	394
6.3.8	File system strutturati a log	398
6.4	ESEMPI DI FILE SYSTEM	399
6.4.1	I file system dei CD-ROM	399
6.4.2	Il file system di CP/M	404
6.4.3	Il file system di MS-DOS	407
6.4.4	Il file system di Windows 98.....	410
6.4.5	Il file system di UNIX V7	413
6.5	RICERCA SUI FILE SYSTEM.....	416
6.6	SOMMARIO	416

7 SISTEMI OPERATIVI MULTIMEDIALI 421

7.1	INTRODUZIONE AI MULTIMEDIA	422
7.2	FILE MULTIMEDIALI	425
7.2.1	Codifica audio	426
7.2.2	Codifica video	428
7.3	COMPRESSEIONE VIDEO.....	430
7.3.1	Lo standard JPEG	431
7.3.2	Lo standard MPEG	433
7.4	SCHEDULAZIONE DEI PROCESSI MULTIMEDIALI	435
7.4.1	Schedulazione di processi omogenei.....	435
7.4.2	Schedulazione generale in tempo reale.....	436
7.4.3	Schedulazione a frequenza monotona.....	437
7.4.4	Schedulazione con priorità alla scadenza più vicina	439
7.5	PARADIGMI PER FILE SYSTEM MULTIMEDIALI.....	440
7.5.1	Funzioni di controllo VCR	441
7.5.2	Video quasi su richiesta	442
7.5.3	Video quasi su richiesta con funzioni VCR	444
7.6	POSIZIONAMENTO DEI FILE	445
7.6.1	Posizionamento di un file su un unico disco.....	446
7.6.2	Due strategie alternative per l'organizzazione dei file.....	446
7.6.3	Posizionare i file per i video quasi su richiesta	450
7.6.4	Posizionare più file su un unico disco.....	451
7.6.5	Posizionamento dei file su dischi diversi	453
7.7	MECCANISMI DI CACHE	455
7.7.1	Meccanismi di cache per i blocchi.....	455
7.7.2	Meccanismi di cache dei file	456
7.8	SCHEDULAZIONE DEL DISCO PER I MULTIMEDIA.....	457
7.8.1	Schedulazione statica del disco	457
7.8.2	Schedulazione dinamica del disco	459

7.9 LA RICERCA SUI MULTIMEDIA	460
7.10 SOMMARIO	461
8 SISTEMI CON PROCESSORI MULTIPLI	465
8.1 MULTIPROCESSORI	467
8.1.1 Hardware multiprocessore	468
8.1.2 Tipi di sistema operativo multiprocessore	474
8.1.3 Sincronizzazione dei multiprocessori	477
8.1.4 Schedulazione dei multiprocessori	481
8.2 MULTICOMPUTER	486
8.2.1 Hardware dei multicompiler	486
8.2.2 Software di comunicazione di basso livello	490
8.2.3 Software di comunicazione a livello utente	492
8.2.4 Chiamate di procedura remote	496
8.2.5 Memoria condivisa distribuita	498
8.2.6 Schedulazione dei multicompiler	502
8.2.7 Bilanciamento del carico	502
8.3 SISTEMI DISTRIBUITI	506
8.3.1 Hardware di rete	508
8.3.2 Servizi e protocolli di rete	510
8.3.3 Middleware basato sui documenti	514
8.3.4 Middleware basato sul file system	515
8.3.5 Middleware basato su oggetti condivisi	521
8.3.6 Middleware basato sulla coordinazione	526
8.4 RICERCA SUI SISTEMI A PROCESSORI MULTIPLI	531
8.5 SOMMARIO	531
9 SICUREZZA	537
9.1 L'AMBIENTE DI SICUREZZA	538
9.1.1 Minacce	538
9.1.2 Intrusi	539
9.1.3 Perdita accidentale di dati	540
9.2 PRINCIPI DI CRITTOGRAFIA	540
9.2.1 Crittografia a chiave privata	541
9.2.2 Crittografia a chiave pubblica	542
9.2.3 Funzioni unidirezionali	543
9.2.4 Firme digitali	543
9.3 AUTENTICAZIONE DELL'UTENTE	544
9.3.1 Autenticazione tramite password	545
9.3.2 Autenticazione tramite oggetti fisici	553

9.3.3 Autenticazione tramite biometrie	555
9.3.4 Contromisure.....	557
9.4 ATTACCHI DALL'INTERNO DEL SISTEMA	558
9.4.1 Cavalli di Troia.....	558
9.4.2 Login spoofing	560
9.4.3 Bombe logiche	561
9.4.4 Porte nascoste	561
9.4.5 Buffer overflow	562
9.4.6 Generici attacchi alla sicurezza.....	564
9.4.7 Famose violazioni della sicurezza.....	565
9.4.8 Principi di progettazione per la sicurezza	566
9.5 ATTACCHI DALL'ESTERNO DEL SISTEMA.....	568
9.5.1 Scenari di danni causati da virus	569
9.5.2 Come agiscono i virus.....	570
9.5.3 Come si diffondono i virus	577
9.5.4 Tecniche antivirus ed anti-antivirus	578
9.5.5 Il Worm di Internet.....	584
9.5.6 Codice mobile.....	586
9.5.7 Sicurezza in Java	591
9.6 MECCANISMI DI PROTEZIONE.....	593
9.6.1 Domini di protezione	593
9.6.2 Liste di controllo degli accessi.....	596
9.6.3 Capability	598
9.7 SISTEMI FIDATI	601
9.7.1 Basi di calcolo fidate (TCB)	602
9.7.2 Modelli formali di sistemi sicuri	603
9.7.3 Sicurezza multilivello	604
9.7.4 L'Orange Book	606
9.7.5 Canali nascosti.....	608
9.8 RICERCHE SULLA SICUREZZA.....	612
9.9 SOMMARIO	612
10 CASO DI STUDIO 1: UNIX E LINUX	619
10.1 LA STORIA DI UNIX.....	620
10.1.1 UNICS.....	620
10.1.2 UNIX del PDP-11	621
10.1.3 UNIX portabile	622
10.1.4 UNIX di Berkeley	623
10.1.5 UNIX standard	623
10.1.6 MINIX	624
10.1.7 Linux	625
10.2 PANORAMICA DI UNIX	627
10.2.1 Obiettivi di UNIX	627
10.2.2 Interfacce con UNIX	628

10.2.3 La shell di UNIX	629
10.2.4 Programmi di utilità di UNIX.....	632
10.2.5 Struttura del kernel	633
10.3 PROCESSI IN UNIX.....	635
10.3.1 Concetti di base	636
10.3.2 Chiamate di sistema per la gestione dei processi in UNIX.....	638
10.3.3 Implementazione dei processi in UNIX.....	643
10.3.4 Avviare UNIX.....	652
10.4 IL MODELLO DI MEMORIA DI UNIX	654
10.4.1 Concetti di base	654
10.4.2 Chiamate di sistema per la gestione della memoria in UNIX.....	657
10.4.3 Implementazione della gestione della memoria in UNIX.....	658
10.5 INPUT/OUTPUT IN UNIX	666
10.5.1 Concetti di base	666
10.5.2 Chiamate di sistema per l'I/O in UNIX.....	668
10.5.3 Implementazione dell'I/O in UNIX	669
10.5.4 Stream.....	672
10.6 IL FILE SYSTEM DI UNIX.....	673
10.6.1 Concetti di base	674
10.6.2 Chiamate di sistema per i file in UNIX	678
10.6.3 Implementazione del file system di UNIX	681
10.6.4 NFS: il file system di rete	687
10.7 LA SICUREZZA IN UNIX.....	692
10.7.1 Concetti di base	692
10.7.2 Chiamate di sistema di sicurezza in UNIX	694
10.7.3 Implementazione della sicurezza in UNIX.....	695
10.8 SOMMARIO	695
11 CASO DI STUDIO 2: WINDOWS 2000	701
11.1 STORIA DI WINDOWS 2000	701
11.1.1 MS-DOS	701
11.1.2 Windows 95/98/Me.....	702
11.1.3 Windows NT	703
11.1.4 Windows 2000	705
11.2 PROGRAMMARE IN WINDOWS 2000.....	709
11.2.1 L'interfaccia di programmazione delle applicazioni (API) Win32...709	
11.2.2 Il registro	711
11.3 LA STRUTTURA DEL SISTEMA	714
11.3.1 La struttura del sistema operativo	715
11.3.2 Implementazione degli oggetti.....	723
11.3.3 Sottosistemi di ambiente.....	728
11.4 I PROCESSI ED I THREAD IN WINDOWS 2000.....	731
11.4.1 Concetti di base	731

11.4.2 Chiamate API per la gestione di job, processi, thread e thread leggeri.....	734
11.4.3 Implementazione dei processi e dei thread.....	737
11.4.4 L'emulazione di MS-DOS.....	742
11.4.5 Avviare Windows 2000	743
11.5 LA GESTIONE DELLA MEMORIA.....	745
11.5.1 Concetti di base.....	745
11.5.2 Le chiamate di sistema per la gestione della memoria.....	749
11.5.3 Implementazione della gestione della memoria.....	750
11.6 INPUT/OUTPUT IN WINDOWS 2000.....	756
11.6.1 Concetti fondamentali.....	756
11.6.2 Chiamate API di Input/Output	757
11.6.3 Implementazione dell'I/O	759
11.6.4 I driver di dispositivi	759
11.7 IL FILE SYSTEM DI WINDOWS 2000.....	762
11.7.1 Concetti fondamentali.....	762
11.7.2 Le chiamate API del file system in Windows 2000	763
11.7.3 Implementazione del file system di Windows 2000.....	764
11.8 LA SICUREZZA IN WINDOWS 2000	775
11.8.1 Concetti fondamentali.....	776
11.8.2 Chiamate API di sicurezza	778
11.8.3 Implementazione della sicurezza	779
11.9 CACHING IN WINDOWS 2000.....	779
11.10 SOMMARIO	781

12 PROGETTAZIONE DI UN SISTEMA OPERATIVO.....785

12.1 LA NATURA DEL PROBLEMA DI PROGETTAZIONE	786
12.1.1 Obiettivi.....	786
12.1.2 Perché è complicato progettare un sistema operativo?	787
12.2 PROGETTAZIONE DELL'INTERFACCIA	789
12.2.1 Principi guida	789
12.2.2 Paradigmi.....	790
12.2.3 L'interfaccia per le chiamate di sistema	794
12.3 IMPLEMENTAZIONE	796
12.3.1 Struttura del sistema	796
12.3.2 Meccanismi contro politiche	799
12.3.3 Ortopogonalità	800
12.3.4 Assegnazione dei nomi	801
12.3.5 Tempo di binding	802
12.3.6 Strutture statiche contro strutture dinamiche	803
12.3.7 Implementazione top-down contro implementazione bottom-up ..	805
12.3.8 Tecniche utili	805
12.4 PRESTAZIONI.....	810
12.4.1 Perché i sistemi operativi sono lenti?	811

12.4.2 Che cosa dovrebbe essere ottimizzato?811
12.4.3 Compromessi tra spazio e tempo.....	.812
12.4.4 L'utilizzo della cache815
12.4.5 Suggerimenti816
12.4.6 Sfruttare la località.....	.816
12.4.7 Ottimizzare il caso comune.....	.817
12.5 GESTIONE DEL PROGETTO817
12.5.1 The Mythical Man Month.....	.817
12.5.2 Struttura del gruppo di lavoro819
12.5.3 Il ruolo dell'esperienza820
12.5.4 Nessuna pallottola è d'argento.....	.821
12.6 TENDENZE NELLA PROGETTAZIONE DI UN SISTEMA OPERATIVO .	.822
12.6.1 Sistemi operativi con un grande spazio di indirizzamento822
12.6.2 Reti823
12.6.3 Sistemi paralleli e distribuiti823
12.6.4 Sistemi multimediali823
12.6.5 Computer alimentati a batteria824
12.6.6 Sistemi embedded824
12.7 SOMMARIO825
13 LETTURE CONSIGLIATE E BIBLIOGRAFIA.....	.829
13.1 SUGGERIMENTI PER ULTERIORI LETTURE829
13.1.1 Introduzione e opere generali.....	.830
13.1.2 Processi e thread830
13.1.3 Deadlock831
13.1.4 Gestione della memoria831
13.1.5 Input/Output831
13.1.6 File system832
13.1.7 Sistemi operativi multimediali832
13.1.8 Sistemi a processori multipli.....	.833
13.1.9 Sicurezza834
13.1.10 UNIX e Linux835
13.1.11 Windows 2000836
13.1.12 Principi di progettazione837
13.2 BIBLIOGRAFIA ALFABETICA837
INDICE ANALITICO863
NOTE SULL'AUTORE875

PREFAZIONE

Il mondo è molto cambiato dalla prima edizione di questo libro, apparsa nel 1992. Reti di computer e sistemi distribuiti di tutti i tipi sono diventati molto comuni; i ragazzini navigano in Internet, mentre in precedenza lo facevano solo gli informatici professionisti. Di conseguenza, anche questo libro è molto cambiato.

Il cambiamento più ovvio consiste nel fatto che la prima edizione era dedicata per circa metà ai sistemi operativi a singolo processore, e per l'altra metà ai sistemi distribuiti: ho scelto questo formato nel 1991, perché allora poche università avevano corsi sui sistemi distribuiti, e tutto ciò che gli studenti imparavano sui sistemi distribuiti doveva essere messo nel corso di sistemi operativi, per il quale questo libro era pensato. Oggi la maggior parte delle università ha un corso separato di sistemi distribuiti, così non è necessario cercare di mettere assieme due argomenti in un solo corso ed un solo libro. Questo libro è pensato per un primo corso di sistemi operativi, e dunque si focalizza maggiormente sui tradizionali sistemi a singolo processore.

Sono stato coautore di altri due libri di sistemi operativi e questo porta a due possibili orientamenti nei corsi:

Orientamento pratico:

1. *Progettazione ed implementazione di sistemi operativi* di Tanenbaum e Woodhull
2. *Sistemi distribuiti* di Tanenbaum e Van Steen

Orientamento tradizionale:

1. *I moderni sistemi operativi* di Tanenbaum
2. *Sistemi distribuiti* di Tanenbaum e Van Steen

Il primo orientamento utilizza MINIX, e gli studenti devono lavorare su MINIX in un laboratorio assistito integrativo del primo corso; il secondo orientamento non utilizza MINIX, ma alcuni piccoli simulatori, che si possono utilizzare per le esercitazioni degli studenti durante un primo corso basato su questo libro. Questi simulatori si possono tro-

vare a partire dalla pagina Web dell'autore, www.cs.vu.nl/~ast/, seguendo il link *Software and Supplementary material for my books*.

Oltre all'enfasi data in questo libro ai sistemi operativi a singolo processore, altri grandi cambiamenti riguardano l'aggiunta di interi capitoli sulla sicurezza dei computer, sistemi operativi multimediali e Windows 2000, tutti argomenti importanti oggigiorno; inoltre è stato aggiunto un nuovo capitolo dedicato alla progettazione dei sistemi operativi, unico nel suo genere.

Un'altra nuova caratteristica è che molti capitoli comprendono anche una sezione dedicata alla ricerca sull'argomento del capitolo: si è pensato in questo modo di introdurre il lettore agli argomenti più attuali sui processi, sulla gestione della memoria e così via. Queste sezioni hanno numerosi riferimenti bibliografici a recenti lavori di ricerca, per coloro che fossero interessati; inoltre il Capitolo 13 ha molti riferimenti a libri introduttivi ed a tutorial.

Infine sono stati aggiunti numerosi argomenti ed altri sono stati ampiamente rivisti. Questi argomenti comprendono: interfacce utente grafiche, sistemi operativi multiprocessori, gestione dell'alimentazione per i portatili, sistemi fidati, virus, terminali di rete, file system di CD-ROM, mutex, RAID, timer software, memorizzazione stabile, scheduling equa e nuovi algoritmi di paginazione. Sono stati aggiunti nuovi problemi ed altri sono stati aggiornati; ora il loro numero totale supera i 450 ed è disponibile un manuale di soluzioni in inglese per i professori che utilizzino questo libro per il loro corso. Inoltre sono stati aggiunti 250 nuovi riferimenti alla letteratura, per aggiornare il libro fino ai giorni nostri.

Nonostante siano state tolte più di 400 pagine di vecchio materiale, il libro ha aumentato le sue dimensioni a causa del nuovo materiale aggiunto: sebbene sia ancora adatto per un corso di un semestre, probabilmente è troppo lungo per un corso di un trimestre nella maggior parte delle università. Per questa ragione, questo libro è stato concepito in maniera modulare: ogni corso di sistemi operativi dovrebbe comprendere i Capitoli dall'1 al 6, poiché queste sono le nozioni fondamentali che uno studente dovrebbe conoscere.

Se vi è ancora del tempo a disposizione, si possono aggiungere altri capitoli, visto che ciascuno di questi presuppone che il lettore abbia terminato i Capitoli dall'1 al 6; i Capitoli dal 7 al 12 sono tutti autocontenuti, perciò se ne può usare un qualsiasi sottoinsieme ed in qualsiasi ordine, a seconda dell'interesse del docente. È opinione dell'autore che i Capitoli dal 7 al 12 siano molto più interessanti dei precedenti, ma sarà compito del docente spiegare agli studenti che dovranno mangiare i broccoli prima di poter gustare la torta al cioccolato.

Vorrei ringraziare le seguenti persone per il loro aiuto nella revisione di parte dei miei manoscritti: Rida Bazzi, Riccardo Bettati, Felipe Cabrera, Richard Chapman, John Connely, John Dickinson, John Elliott, Deborah Frincke, Chandana Gamage, Robbert Geist, David Golds, Jim Griffioen, Gary Harkin, Frank Kaashoek, Mukkai Krishnamoorthy, Monica Lam, Jussi Leiwo, Herb Mayer, Kirk McKusick, Evi Nemeth, Bill Potvin, Prasant Shenoy, Thomas Skinner, Xian-He Sun, William Terry, Robbert Van Renesse e Maarten Van Steen. Jamie Hanrahan, Mark Russinovich e Dave Solomon, profondi conoscitori di Windows 2000, sono stati di molto aiuto. Un ringraziamento speciale va a Al Woodhull per le sue preziose revisioni e il suo aiuto nel pensare molti nuovi problemi di fine capitolo.

Anche i miei studenti sono stati di grande aiuto, con commenti e feedback, specialmente Staas de Jong, Jan de Vos, Niels Drost, David Fokkema, Auke Folkerts, Peter Groenewegen, Wilco Ibes, Stefan Jansen, Jeroen Ketema, Joeri Mulder, Irwin Oppenheim, Stef Post, Umar Rehman, Daniel Rijkhof, Maarten Sander, Maurits van der Schee, Rik van der Stoel, Mark van Driel, Dennis van Veen e Thomas Zeeman.

Barbara e Marvin sono ancora meravigliosi, come al solito, ognuno in un modo unico. Infine, ma non perché sia l'ultima, vorrei ringraziare Suzanne per il suo amore e per la sua pazienza, per non menzionare tutti i *druiven* e *kersen*, che negli ultimi tempi hanno preso il posto del *sinasappelsap*.

PREFAZIONE ALL'EDIZIONE ITALIANA

È semplice presentare la seconda edizione di uno dei testi più popolari sui sistemi operativi, scritto da Andrew Tanenbaum, professore alla Vrije Universiteit di Amsterdam, che ha ricevuto premi prestigiosi per l'attività didattica e scientifica e ha progettato diversi ben noti sistemi operativi. Basta dire che questo libro, appena uscito, è già un classico.

Pensato per un corso universitario di introduzione ai sistemi operativi, il libro contiene una esposizione sintetica, chiara e precisa dei principi fondamentali, corredata da un vasto numero di esempi dettagliati; ogni capitolo arriva a coprire anche tecniche e strumenti innovativi, ed è completato da una visione d'insieme degli argomenti di ricerca più attuali. Il libro perciò non esaurisce la propria utilità con il superamento dell'esame universitario, ma può accompagnare validamente l'intero percorso degli studi, e rappresentare un momento di aggiornamento professionalmente significativo per chi già si trova nel mondo del lavoro.

In particolare, questo libro contiene due capitoli dedicati rispettivamente a UNIX e Windows 2000, i due mondi che rappresentano la quasi totalità dei sistemi operativi su personal computer, ma anche su workstation e server, con i quali gli studenti di oggi dovranno necessariamente avere a che fare sia nei laboratori universitari, sia nell'attività professionale.

Solo chi ha l'esperienza e l'autorevolezza di Tanenbaum può poi inserire un capitolo dedicato ai principi di progettazione dei sistemi operativi, un concentrato dell'esperienza di una vita, che risulta utile leggere, e rileggere, anche a chi non è più uno studente dei primi anni di università.

Il tutto scritto con uno stile molto personale, che rende vivaci anche le descrizioni più tecniche, riuscendo ad ottenere una efficacia didattica superiore: l'impressione lasciata dalla lettura di una pagina spesso è quella di ascoltare una vera e propria lezione universitaria.

INTRODUZIONE

Un calcolatore moderno è costituito da uno o più processori, da un po' di memoria principale, dischi, stampanti, una tastiera, un video, interfacce di rete e altri dispositivi di ingresso/uscita: in definitiva, è un sistema complesso. Scrivere programmi che tengano conto di tutte queste componenti e che le usino correttamente, anche se non in modo ottimale, è estremamente difficile. Per questo motivo, i calcolatori vengono dotati di uno strato di software detto **sistema operativo**, il cui compito è gestire tutti questi dispositivi e fornire ai programmi utente un'interfaccia semplificata con l'hardware. I sistemi operativi sono l'argomento di questo libro.

La Figura 1.1 descrive la collocazione del sistema operativo rispetto all'architettura di un calcolatore: al livello inferiore abbiamo l'hardware, che, in molti casi, è a sua volta composto di due o più livelli (o strati). Il livello più basso contiene i dispositivi fisici, composti da circuiti integrati, cavi, alimentatori, tubi a raggi catodici e altri dispositivi simili. Come questi dispositivi siano realizzati e come operino è competenza degli ingegneri elettronici.

Salendo abbiamo il **livello di microarchitettura**, nel quale i dispositivi fisici sono raggruppati a formare unità funzionali. Tipicamente, questo livello contiene qualche registro interno alla CPU (Central Processing Unit – Unità Centrale di Elaborazione) e un data path, contenente un'unità aritmetico logica. Ad ogni ciclo di clock, uno o due operandi vengono prelevati dai registri e combinati nell'unità aritmetico logica (per esempio, addizionandoli o combinandoli con l'operatore booleano AND). Il risultato dell'elaborazione viene poi memorizzato in uno o più registri. Su alcune macchine, l'operazione di data path è controllata da software apposito, il **micropogramma**, mentre su altre quest'operazione viene effettuata direttamente dai circuiti hardware.

Lo scopo del data path è eseguire insiemi di istruzioni, alcune delle quali vengono completate in un ciclo di data path, mentre altre possono richiederne più d'uno; queste istruzioni possono utilizzare registri o altri dispositivi hardware. L'insieme di hardware e istruzioni a disposizione di un programmatore in linguaggio assembly forma il **livello ISA (Instruction Set Architecture)**, spesso chiamato **linguaggio macchina**.

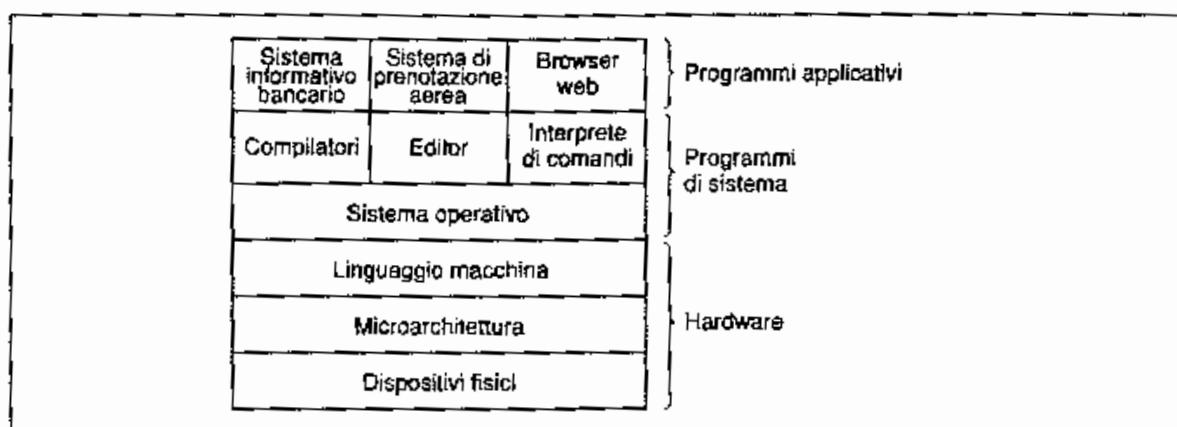


Figura 1.1 Un calcolatore è costituito da hardware, programmi di sistema e programmi applicativi

Il linguaggio macchina comprende tipicamente tra le 50 e le 300 istruzioni che, per la maggior parte, hanno lo scopo di spostare dati all'interno della macchina, eseguire calcoli aritmetici e confrontare valori. A questo livello, i dispositivi di ingresso/uscita sono controllati mediante il caricamento di valori in speciali registri di dispositivo: per esempio, è possibile ordinare ad un disco di eseguire una lettura caricando nei suoi registri di dispositivo il valore dell'indirizzo su disco, l'indirizzo in memoria principale, numero di byte e direzione (lettura o scrittura). In pratica occorrono molti altri parametri, e lo stato restituito dal disco dopo l'esecuzione di un'operazione è molto complesso. Inoltre, la temporizzazione dei comandi gioca un ruolo importante nella programmazione di molti dispositivi di ingresso/uscita.

Per mascherare questa complessità, viene fornito un sistema operativo, costituito da uno strato di software che nasconde parzialmente l'hardware e dà al programmatore un insieme di istruzioni più pratico con cui lavorare. Per esempio, utilizzare l'istruzione `read block` da un file è concettualmente più semplice che preoccuparsi di dettagli quali muovere le testine del disco, aspettare che siano in posizione corretta e così via.

La parte restante del software di sistema sta sopra il sistema operativo: qui troviamo l'interprete dei comandi (shell), i sistemi a finestre, i compilatori, gli editor e gli altri programmi indipendenti dalle applicazioni. È importante capire che questi programmi non fanno assolutamente parte del sistema operativo, anche se vengono generalmente forniti da chi produce il calcolatore: è un punto cruciale, ma sottile. Il sistema operativo è (normalmente) quella porzione di software che viene eseguito in modalità **kernel** (kernel mode) o modalità **supervisore** (supervisor mode), ed è protetto dalle manomissioni degli utenti dall'hardware (per il momento iguoriamo il fatto che alcuni microprocessori più vecchi o di categoria inferiore non mettano a disposizione alcun meccanismo di protezione hardware). I compilatori e gli editor vengono eseguiti in modalità **utente** (user mode). Se un utente non gradisce un certo compilatore, egli ("egli" va inteso come "egli o ella", per tutto il libro) è libero di scriversi il proprio, se vuole, ma non è libero di scrivere il proprio gestore delle interruzioni, che è parte del sistema operativo ed è normalmente protetto con meccanismi hardware da ogni tentativo di modifica da parte dell'utente.

Questa distinzione, comunque, può risultare sfocata nei sistemi embedded, che possono non avere la modalità kernel, o nei sistemi interpretati, quali sistemi operativi basati su Java, che usano l'interpretazione, non l'hardware, per distinguere le componenti. Nondimeno, per i calcolatori tradizionali, il sistema operativo è ciò che viene eseguito in modalità kernel.

Detto questo, in molti sistemi ci sono programmi che vengono eseguiti in modalità

utente, ma che aiutano il sistema operativo o hanno funzioni privilegiate. Per esempio, c'è spesso un programma che permette agli utenti di cambiare le proprie password, che non fa parte del sistema operativo e non viene eseguito in modalità kernel, ma, evidentemente, svolge una funzione delicata e deve essere salvaguardato in modo particolare.

In qualche sistema, quest'idea viene portata all'estremo e una parte di quanto viene normalmente considerato sistema operativo, come il file system, viene eseguita in modalità utente. In sistemi come questi, è difficile tracciare un confine netto: tutto ciò che viene eseguito in modalità kernel fa ovviamente parte del sistema operativo, ma è possibile che alcuni programmi che non vengono eseguiti in questa modalità siano parte di esso o, almeno, gli siano strettamente collegati.

Infine, al di sopra dei programmi di sistema, abbiamo le applicazioni: questi programmi vengono acquistati, o scritti dagli utenti, per risolvere i propri problemi specifici, come, ad esempio, elaborazione di testi, utilizzo di fogli di calcolo, esecuzione di calcoli d'ingegneria o memorizzazione di informazioni in una base di dati.

1.1 Che cos'è un sistema operativo?

La maggior parte degli utenti di calcolatori ha una qualche esperienza di sistemi operativi, tuttavia risulta difficile definire esattamente cosa sia un sistema operativo. Parte del problema sta nel fatto che il sistema operativo realizza due funzionalità che sono praticamente scorrelate, estendendo la macchina e gestendo risorse e, a seconda di chi parla, sentirete dire più cose dell'una o dell'altra. Diamo un'occhiata ad entrambe queste funzionalità.

1.1.1 Il sistema operativo come macchina estesa

Come già accennato, l'architettura (insieme di istruzioni, organizzazione della memoria, ingresso/uscita e struttura del bus) della maggior parte dei calcolatori, a livello di linguaggio macchina è primitiva e risulta difficile da programmare, soprattutto per quello che riguarda l'ingresso/uscita. Per rendere più concreta questa considerazione, diamo un'occhiata a come viene realizzato l'ingresso/uscita di un floppy disk usando il chip di controllo NEC PD765 compatibile, che viene impiegato dalla maggior parte dei personal computer Intel. (In tutto il libro, useremo indifferentemente i termini "dischetto" e "floppy disk".) Il PD765 ha 16 comandi, ognuno dei quali specificato caricando da 1 a 9 byte in un registro di dispositivo, che servono a leggere e scrivere dati, a muovere il braccio delle testine e a formattare tracce, così come a inizializzare, rilevare lo stato, riportare nelle condizioni iniziali e ricalibrare il controllore e i dischetti.

I comandi più elementari sono `read` e `write`, ognuno dei quali richiede 13 parametri, raggruppati in 9 byte. Questi parametri specificano informazioni quali l'indirizzo del blocco del dischetto da leggere, il numero di settori per traccia, la modalità di registrazione delle informazioni sul supporto fisico, la distanza tra i settori e cosa fare quando si incontra un contrassegno di dato cancellato. Se non capite qualcosa di questo gergo, non vi preoccupate: è esattamente questo il punto: è piuttosto esoterico. Quando l'operazione viene completata, il controllore del dischetto restituisce 23 campi di errore o di stato, raggruppati in 7 byte. Come se non bastasse, chi programma il floppy disk deve tenere costantemente sotto controllo lo stato di accensione del motore: se è spento, deve essere riacceso, il che comporta un lungo ritardo, prima che i dati possano essere letti o scritti. Il motore non può comunque restare acceso molto a lungo, altrimenti il dischetto si

potrebbe consumare. Di conseguenza, il programmatore è costretto a trovare un compromesso fra il lungo ritardo dell'avvio del motore e l'usura dei floppy disk, che comporta la perdita dei dati che contengono.

Senza andare nei dettagli *reali*, dovrebbe essere chiaro che il programmatore medio, probabilmente, non vuole essere troppo coinvolto nella programmazione dei floppy disk o dei dischi rigidi (*hard disk*), che sono altrettanto complessi anche se abbastanza diversi, mentre vuole avere a che fare con una semplice astrazione ad alto livello. Nel caso dei dischi, una tipica astrazione consiste nel vedere il disco come una collezione di file, ciascuno con un nome; ogni file può essere aperto in lettura o scrittura, quindi letto o scritto ed infine può essere chiuso. Dettagli quali il fatto che venga usata o meno la modulazione di frequenza per la scrittura dei dati e quale sia lo stato del motore non dovrebbero apparire nell'astrazione presentata all'utente.

Naturalmente, il programma che nasconde al programmatore la verità sull'hardware e gli presenta una graziosa e semplice serie di file con nome che possono essere letti e scritti è, ancora una volta, il sistema operativo. Così come maschera i dettagli dell'hardware del disco e presenta invece al programmatore una semplice interfaccia a file, il sistema operativo gli cela tutta una serie di spiacevoli faccende che riguardano le interruzioni, i timer, la gestione della memoria ed altre caratteristiche a basso livello. In ogni caso, l'astrazione offerta dal sistema operativo è più semplice e più facile da usare rispetto a quella fornita dall'hardware sottostante.

In questo senso, la funzionalità del sistema operativo è quella di presentare all'utente una macchina estesa o una macchina virtuale, che sia più facile da programmare dell'hardware sottostante. Vedere come il sistema operativo riesca a raggiungere questo obiettivo rappresenta una lunga storia, ed è esattamente ciò che studieremo in dettaglio in tutto questo libro. Per riassumere il concetto, il sistema operativo fornisce una varietà di servizi, di cui i programmi possono usufruire attraverso istruzioni speciali dette chiamate di sistema (*system call*). Analizzeremo alcune delle chiamate di sistema più comuni, più avanti in questo capitolo.

1.1.2 Il sistema operativo come gestore delle risorse

Il concetto di sistema operativo come entità che ha come compito principale quello di mettere a disposizione dell'utente una comoda interfaccia corrisponde a guardare le cose dall'alto verso il basso. Guardando le cose dal basso verso l'alto, si arriva alla conclusione che il sistema operativo sia stato messo lì per controllare e gestire tutti i pezzi di un sistema complesso. I calcolatori moderni sono costituiti da processori, memorie, timer, dischi, mouse, interfacce di rete, stampanti e una gran varietà di altri dispositivi. Da un punto di vista alternativo, il compito del sistema operativo è quello di gestire un'allocazione ordinata di queste risorse ai vari programmi che competono tra loro per usarle.

Immaginate cosa succederebbe se tre programmi che sono in esecuzione su un calcolatore tentassero contemporaneamente di usare la stessa stampante: le prime linee di stampa potrebbero provenire dal programma 1, successivamente alcune righe dal programma 2, quindi alcune dal programma 3 e così via e il risultato sarebbe il caos. Il sistema operativo può risolvere una simile situazione di potenziale caos bufferizzando (cioè salvando temporaneamente) i dati destinati alla stampante sul disco. Quando un programma termina, il sistema operativo può copiare le uscite di questo programma dal file su disco dove sono state memorizzate alla stampante, mentre gli altri programmi possono continuare a generare altre uscite, senza sapere che queste non vengono (ancora) realmente indirizzate alla stampante.

Quando un calcolatore, o una rete, vengono utilizzati da più utenti, la necessità di

gestire e proteggere la memoria, i dispositivi di ingresso/uscita e le altre risorse è ancora più evidente, dal momento che altrimenti gli utenti potrebbero interferire tra di loro; inoltre, gli utenti hanno spesso la necessità di condividere non solo l'hardware, ma anche informazioni (file, basi di dati, etc.). In breve, questa visione del sistema operativo racchiude l'idea che il suo primo compito sia quello di tenere traccia di chi stia usando quale risorsa, soddisfare le richieste delle risorse, contabilizzarne l'uso e mediare richieste conflittuali provenienti da programmi e utenti diversi.

La gestione delle risorse comporta la loro condivisione sotto due aspetti: rispetto al tempo e rispetto allo spazio. Quando una risorsa è condivisa rispetto al tempo, programmi o utenti diversi fanno a turno per usarla: prima uno di questi usa la risorsa, poi un altro e così via. Per esempio, con un solo processore e più programmi che devono essere eseguiti su di esso, il sistema operativo prima alloca il processore a un programma; quando questo è stato in esecuzione per un tempo sufficiente, un altro programma prende in uso il processore, poi un altro e quindi, eventualmente, di nuovo il primo. Determinare come la risorsa sia condivisa rispetto al tempo (quale sia il processo successivo e per quanto tempo debba utilizzare la risorsa) è compito del sistema operativo. Un altro esempio di condivisione rispetto al tempo è la condivisione della stampante: quando più job di stampa vengono accodati per la stampa su una singola stampante, occorre decidere quale sia il job successivo da stampare.

L'altro tipo di condivisione è rispetto allo spazio: invece di alternarsi, ad ognuno viene assegnata parte della risorsa. Per esempio, la memoria principale è normalmente ripartita fra diversi programmi in esecuzione, così che possano stare in memoria contemporaneamente (per esempio, allo scopo di fare a turno per l'uso della CPU). Supponendo che ci sia abbastanza memoria per più programmi, è più efficiente mantenerli in memoria tutti insieme, piuttosto che assegnare tutta la memoria a uno di essi, in particolare se questo necessita solo di una piccola porzione di essa. Indubbiamente, questo crea problemi di equità, protezione e così via, ed è dovere del sistema operativo risolverli. Un'altra risorsa che è condivisa rispetto allo spazio è il disco (rigido); in molti sistemi, un disco può contenere contemporaneamente i file di molti utenti, per cui l'allocazione dello spazio su disco e il tenere traccia di chi stia usando uno specifico blocco è un compito tipico di gestione di risorse del sistema operativo.

1.2 Storia dei sistemi operativi

I sistemi operativi si sono evoluti nel corso degli anni e, nelle prossime sezioni, daremo una rapida occhiata ai momenti salienti della loro storia. Dal momento che, storicamente, i sistemi operativi sono stati strettamente legati all'architettura dei calcolatori sui quali erano in esecuzione, vedremo diverse generazioni di calcolatori, per capire quali sistemi operativi utilizzassero. Questa corrispondenza fra generazioni di sistemi operativi e di macchine può essere considerata brutale, ma darà all'argomento una struttura che altrimenti non sarebbe possibile.

Il primo autentico calcolatore digitale fu progettato dal matematico inglese Charles Babbage (1792-1871). Sebbene Babbage abbia speso gran parte della propria vita e della propria fortuna cercando di costruire questa "macchina analitica", egli non riuscì mai a vederla funzionare in maniera corretta, dal momento che fu progettata come dispositivo meccanico e che la tecnologia dell'epoca non era in grado di produrre le ruote, gli ingranaggi e le altre parti meccaniche necessarie con la precisione richiesta. Inutile dire che la macchina analitica non aveva un sistema operativo.

Curiosità storica, Babbage capì che sarebbe stato necessario del software per la sua

macchina analitica, così assunse una giovane donna, Ada Lovelace, figlia del noto poeta britannico Lord Byron, come prima programmatrice del mondo. Il linguaggio di programmazione Ada[®] prende da lei il suo nome.

1.2.1 La prima generazione (1945-1955): valvole e schede a spinotti

Dopo gli inutili sforzi di Babbage, furono fatti pochi progressi nella realizzazione di calcolatori digitali fino alla seconda guerra mondiale. Circa verso la metà degli anni 40, Howard Aiken ad Harvard, John von Neumann all'Institute for Advanced Study di Princeton, J. Presper Eckert e William Mauchley all'Università della Pennsylvania e Konrad Zuse in Germania, fra gli altri, riuscirono a costruire macchine da calcolo. Le prime usavano relè meccanici ma erano molto lente, con tempi di ciclo misurabili in secondi. I relè furono più tardi sostituiti da valvole termoioniche. Queste macchine erano enormi, riempivano intere stanze con centinaia di migliaia di valvole, ma erano milioni di volte più lente del più economico degli attuali personal computer.

In quei primi giorni, un singolo gruppo di persone progettava, costruiva, programmava, faceva lavorare ed eseguiva la manutenzione di ciascuna macchina. Tutta la programmazione veniva totalmente effettuata in linguaggio macchina, spesso predisponendo una serie di spinotti su schede particolari che servivano per controllare le funzioni più elementari della macchina. I linguaggi di programmazione, linguaggio assembler compreso, erano sconosciuti e di sistemi operativi non si sentiva nemmeno parlare. Normalmente, le cose si svolgevano in questo modo: il programmatore si registrava su un foglio delle prenotazioni appeso al muro per un certo numero di ore, poi andava nella sala macchine, inseriva la propria scheda a spinotti con il programma nel calcolatore e passava le ore successive a sperare che nessuna delle circa 20000 valvole si bruciasse mentre il programma girava. La quasi totalità delle applicazioni erano semplici calcoli numerici, come produzioni di tabelle di seni, coseni e logaritmi.

Negli anni '50 questa procedura venne migliorata con l'introduzione delle schede perforate. Ora era possibile scrivere programmi sulle schede e leggerli tramite il calcolatore, invece di usare le schede a spinotti; per il resto, la procedura era assolutamente la stessa.

1.2.2 La seconda generazione (1955-1965): transistor e sistemi batch

L'introduzione dei transistor durante la metà degli anni '50 cambiò radicalmente la situazione. I calcolatori divennero abbastanza affidabili da poter essere costruiti e venduti dando all'acquirente la ragionevole certezza che avrebbero funzionato abbastanza a lungo da poter svolgere un lavoro utile e, per la prima volta, ci fu una chiara separazione fra progettisti, costruttori, operatori, programmatore e personale di manutenzione.

Questi calcolatori, ora chiamati mainframe, erano tenuti chiusi in speciali sale macchine con aria condizionata, ed al loro funzionamento erano assegnati gruppi di operatori specializzati. Il loro costo, di molti milioni di dollari, poteva essere affrontato solo da grosse compagnie, dalle agenzie governative o dalle università. Per far girare un job (cioè un programma o un insieme di programmi), un programmatore doveva prima scrivere il programma su carta (in FORTRAN o in assembler), poi doveva copiarlo su schede perforate, infine doveva portare il pacchetto delle schede nella stanza dove venivano raccolti i

programmi e darlo ad uno degli operatori. Dopodiché poteva andare a prendersi un caffè in attesa dei risultati in uscita.

Appena il calcolatore aveva terminato il job in esecuzione in quel momento, qualsiasi cosa fosse, un operatore andava alla stampante, prelevava quanto prodotto in uscita dal job e lo portava nella stanza dove venivano distribuiti i risultati delle elaborazioni, in modo che il programmatore lo potesse ritirare più tardi; quindi prendeva uno dei pacchetti di schede dalla stanza di raccolta dei programmi e lo faceva leggere al calcolatore. Se risultava necessario anche il compilatore FORTRAN, l'operatore lo doveva prendere dall'archivio e farlo leggere al calcolatore. Gran parte del tempo di elaborazione andava quindi sprecato mentre gli operatori andavano avanti e indietro per la sala macchine.

Dato l'elevato costo delle macchine, è facile capire come la gente si mettesse rapidamente a cercare di ridurre il tempo sprecato: la soluzione che venne adottata nella maggior parte dei casi fu quella dei sistemi batch (sistemi di elaborazione a lotti). L'idea era quella di raccogliere un intero cassetto di job nella stanza dedicata alla raccolta dei programmi e di farli trasferire su nastro magnetico da un piccolo calcolatore (relativamente) poco costoso, come ad esempio un IBM 1401, molto adatto a leggere schede perforate, scrivere nastri magnetici o stampare dati, ma non adatto ad eseguire calcoli. Per eseguire i calcoli venivano usati altri calcolatori, molto più costosi, come gli IBM 7094: questa situazione è illustrata in Figura 1.2.

Dopo circa un'ora durante la quale veniva raccolto un batch (lotto) di lavori, il nastro veniva riavvolto e portato nella sala macchine, dove veniva montato sul lettore di nastri. L'operatore caricava quindi un programma speciale (l'antenato degli attuali sistemi operativi) che leggeva il primo job e lo eseguiva; i dati in uscita, anziché venire stampati, venivano registrati su un secondo nastro magnetico. Dopo la fine di ognuno dei job, il sistema operativo leggeva in maniera automatica il prossimo job dal nastro di ingresso e lo mandava in esecuzione. Quando l'intero lotto di lavori era stato eseguito, l'operatore rimuoveva i nastri di ingresso e di uscita, li sostituiva con quelli di un nuovo lotto di lavori e portava il nastro di uscita al 1401 per stamparne i dati fuori linea (off line), cioè senza essere direttamente collegato al calcolatore principale.

La struttura tipica di un job è presentata in Figura 1.3: esso cominciava con una scheda di tipo \$JOB, che specificava il tempo di esecuzione massimo previsto (in minuti), il numero del conto su cui doveva venire addebitato il costo dell'elaborazione, ed il nome del programmatore. Poi c'era una scheda di tipo \$FORTRAN, che richiedeva al sistema operativo di caricare il compilatore FORTRAN dal nastro di sistema; a questa scheda

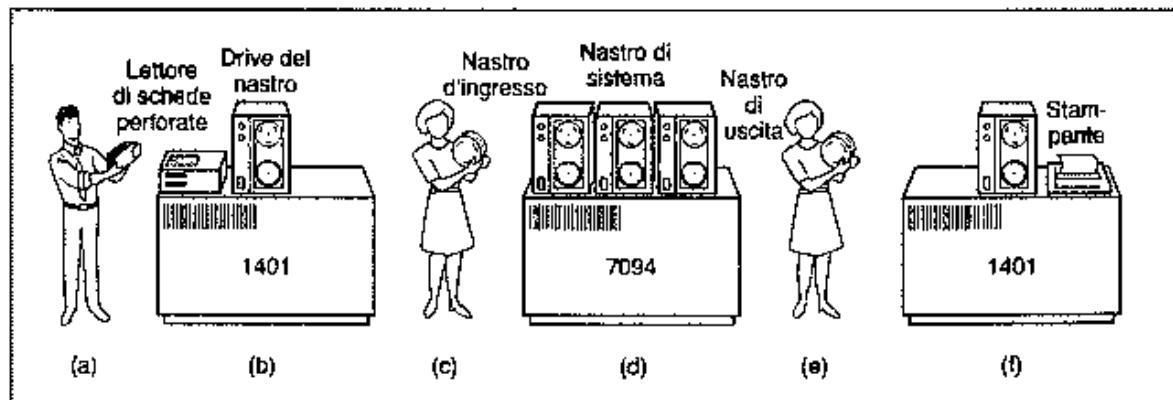


Figura 1.2 Uno dei primi sistemi di elaborazione batch. (a) Il programmatore porta le schede perforate al 1401. (b) Il 1401 legge un lotto di programmi sul nastro. (c) L'operatore porta un nastro d'ingresso al 7094. (d) Il 7094 esegue i calcoli. (e) L'operatore porta il nastro in uscita al 1401. (f) Il 1401 stampa l'uscita.

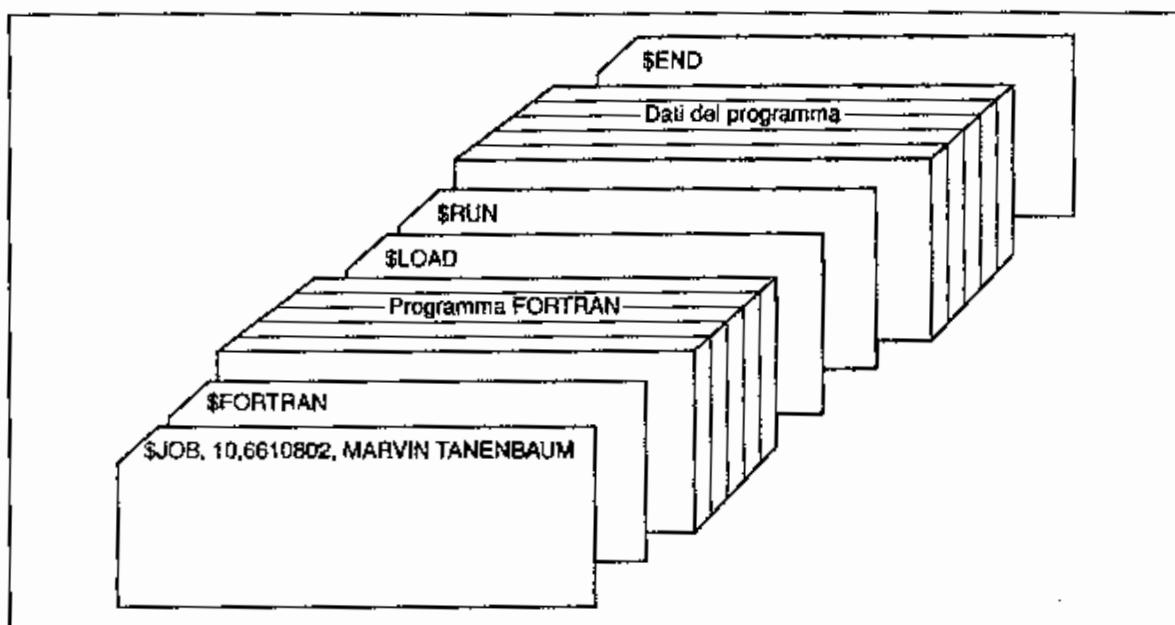


Figura 1.3 Struttura di un tipico job FMS.

faceva seguito il programma da compilare, quindi una scheda di tipo \$LOAD che comunicava al sistema operativo che il programma oggetto appena compilato doveva essere caricato per l'esecuzione (i programmi oggetto prodotti dalla compilazione venivano spesso scritti su nastri di servizio e dovevano essere caricati in maniera esplicita). C'era poi una scheda di tipo \$RUN, che richiedeva al sistema operativo di mandare in esecuzione il programma con i dati contenuti sulle schede immediatamente seguenti. Infine, una scheda di tipo \$END contrassegnava la fine del job. Queste schede di controllo un po' primitive costituivano i precursori dei moderni linguaggi di controllo dei job e degli interpreti di comandi.

I grandi calcolatori della seconda generazione furono principalmente usati per calcoli scientifici ed ingegneristici, come per esempio nella risoluzione di sistemi di equazioni differenziali alle derivate parziali. Per lo più furono programmati in FORTRAN ed in linguaggio assembler. I sistemi operativi tipici furono FMS (Fortran Monitor System) e IBSYS, sistema operativo dell'IBM per 7094.

1.2.3 La terza generazione (1965-1980): circuiti integrati e multiprogrammazione

All'inizio degli anni '60, la maggior parte dei produttori di calcolatori aveva due linee distinte di prodotti, completamente incompatibili fra di loro: da una parte c'erano i grossi calcolatori scientifici orientati alla parola, come l'IBM 7094, che venivano usati per calcoli numerici scientifici ed ingegneristici; dall'altra, c'erano i calcolatori commerciali, orientati al carattere, come l'IBM 1401, che venivano largamente usati dalle ditte di assicurazioni e dalle banche per ordinare dati su nastro o per le stampe.

Sviluppare e mantenere due distinte linee di prodotti era un'attività molto dispendiosa per le ditte che costruivano calcolatori; inoltre, molti nuovi clienti richiedevano inizialmente una macchina piccola, ma in seguito arrivavano a saturarla completamente e ne richiedevano un'altra, che potesse comunque far girare tutti i loro programmi, ma molto più velocemente.

L'IBM tentò di risolvere tutti e due questi problemi in un colpo solo con l'introduzione del Sistema 360 (System/360). Il Sistema 360 era costituito da una serie di macchine, compatibili fra di loro a livello software, che andavano da macchine piccole, tipo 1401, a macchine molto più potenti del 7094. Queste macchine differivano unicamente nel prezzo e nelle prestazioni (quantità massima di memoria, velocità del processore, numero di dispositivi di ingresso/uscita collegabili, eccetera). Dal momento che tutte le macchine avevano la stessa architettura e lo stesso insieme d'istruzioni, almeno in teoria i programmi scritti per una macchina potevano girare su tutte le altre; per di più, il Sistema 360 fu progettato per calcoli sia di tipo commerciale sia di tipo scientifico, cioè numerico. In questo modo, una singola famiglia di calcolatori poteva soddisfare le esigenze di tutti i clienti. Negli anni successivi, IBM rese disponibili calcolatori con tecnologia più moderna, che mantenevano la compatibilità con la famiglia 360, conosciuti come serie 370, 4300, 3080 e 3090.

Il Sistema 360 fu la prima linea di calcolatori ad usare la tecnologia dei circuiti integrati, allora a bassa scala di integrazione, che portò vantaggi rilevanti, in termini di prezzo e di prestazioni, rispetto alla seconda generazione di macchine, costruite a partire da singoli transistor. Fu un successo immediato e l'idea della famiglia di calcolatori compatibili fu immediatamente adottata dagli altri principali costruttori di calcolatori. I discendenti di queste macchine sono ancora usati nei grossi centri di calcolo dei nostri giorni; oggi giorno, gestiscono enormi basi di dati (ad esempio, i sistemi di prenotazione delle linee aeree) o sono i server dei siti World Wide Web che devono elaborare migliaia di richieste al secondo.

L'idea di avere una singola famiglia di macchine fu contemporaneamente una grande forza, ma anche una grande debolezza. L'intenzione era che tutto il software, compreso il sistema operativo, OS/360, dovesse girare su tutti i modelli: sui piccoli sistemi, che spesso sostituirono i 1401 per copiare le schede su nastro, e su sistemi molto grossi, che sostituirono spesso i 7094 per le previsioni del tempo o altri compiti di calcolo molto gravosi; doveva funzionare correttamente su sistemi con poche periferiche come su sistemi con molte periferiche; doveva lavorare in ambiente scientifico e in ambiente commerciale; soprattutto, doveva essere efficiente in tutti questi compiti.

Non c'era nessun modo, né per IBM né per altri, di scrivere un software che rispettasse tutti questi requisiti in conflitto fra di loro: il risultato fu un sistema operativo enorme e straordinariamente complesso, probabilmente da due a tre ordini di grandezza più grande di FMS. Esso era composto da milioni di righe di codice assembler scritte da migliaia di programmatore e conteneva migliaia di errori, che resero necessaria una successione di nuove versioni per cercare di correggerli. Ciascuna nuova versione corregeva alcuni errori, ma ne introduceva altri, così che il numero di errori rimase probabilmente costante nel tempo.

Uno dei progettisti di OS/360, Fred Brooks, scrisse in seguito un libro incisivo ed umoristico (Brooks, 1996) che descriveva la propria esperienza con OS/360. Nonostante non sia possibile riassumerlo qui, basti dire che la copertina mostra un branco di bestie preistoriche invischiate in una pozza di catrame. La copertina del libro di Silberschatz ed altri (2000) sottolinea un concetto analogo, con sistemi operativi rappresentati da dinosauri.

Nonostante l'enormità delle dimensioni e dei problemi, OS/360 e gli altri sistemi operativi della terza generazione prodotti da altri costruttori di calcolatori soddisfecero, in pratica, la maggioranza dei loro clienti. Essi resero anche popolari alcune tecniche chiave che non erano presenti nei sistemi della seconda generazione, la più importante delle quali, probabilmente, è la multiprogrammazione. Sul 7094, quando il job corrente era fermo in attesa del completamento di un'operazione su nastro o su qualche altro dispositivo di ingresso/uscita, la CPU rimaneva inattiva fino al completamento dell'attività di

ingresso/uscita. Nel caso di calcoli scientifici intensivi (CPU-bound) le attività di ingresso/uscita sono relativamente infrequenti e di conseguenza le perdite di tempo non sono significative; viceversa, nel caso di elaborazioni commerciali l'attività di ingresso/uscita può spesso arrivare all'80 o 90 per cento del tempo totale di elaborazione, cosicché risultò necessario fare qualcosa per evitare che la (costosa) CPU rimanesse inattiva per un tempo così lungo.

La soluzione sviluppata consisteva nel dividere la memoria centrale in alcune partizioni, con un job diverso per ogni partizione, come illustrato dalla Figura 1.4: mentre un job rimaneva in attesa del completamento di una certa attività di ingresso/uscita, la CPU poteva essere usata da un altro job e quando si riusciva a tenere in memoria un numero sufficiente di job contemporaneamente, si riusciva a mantenere occupata la CPU quasi per il 100 per cento del tempo. Mantenere con sicurezza un certo insieme di job in memoria centrale richiede la presenza di hardware specializzato per proteggere ciascun job da "sbirciate" o interferenze da parte degli altri job, ma i 360 e gli altri sistemi della terza generazione disponevano di questi particolari meccanismi hardware.

Un'altra caratteristica importante dei sistemi operativi della terza generazione era quella di permettere la lettura dei job dalle schede al disco non appena le schede venivano portate alla sala macchine. Poi, non appena il job in esecuzione terminava, il sistema operativo poteva caricare un nuovo job dal disco nella partizione di memoria che si era resa disponibile e mandarlo in esecuzione. Questa tecnica viene chiamata *spooling* (Simultaneous Peripheral Operation On Line, operazioni simultanee sulla periferica in linea) e fu usata anche per gestire l'uscita dei risultati: con lo spooling, i 1401 non erano più necessari e sparì anche gran parte dell'attività di spostamento dei nastri magnetici.

Sebbene i sistemi operativi della terza generazione fossero adatti alle grandi elaborazioni scientifiche ed alle elaborazioni di carattere commerciale, essi erano ancora in buona sostanza sistemi batch. Molti programmatore rimasero legati ai giorni della prima generazione, quando avevano la macchina a completa disposizione per qualche ora e, di conseguenza, potevano mettere a punto i loro programmi molto velocemente; con i sistemi della terza generazione, il tempo che intercorreva fra il momento in cui il job veniva portato in sala macchine e il momento in cui potevano essere ritirati i risultati era spesso di alcune ore. Di conseguenza, un singolo errore nel posizionamento di una virgola poteva causare il fallimento della compilazione e, in particolare, poteva far perdere un'intera inezza giornata al programmatore.

Il desiderio di un tempo di risposta veloce preparò la strada al *timesharing* (condivisione di tempo), una variante della multiprogrammazione nella quale ogni utente aveva a disposizione un terminale in linea. Nei sistemi timesharing, se 20 utenti erano collegati contemporaneamente e 17 di loro stavano pensando, bevendo caffè o parlando, la CPU poteva essere assegnata all'esecuzione a turno dei tre job che in quel momento richiedevano il servizio. Dal momento che chi mette a punto i programmi in genere richiede l'esecuzione di comandi brevi, come, ad esempio, "compila una procedura (per tutto il

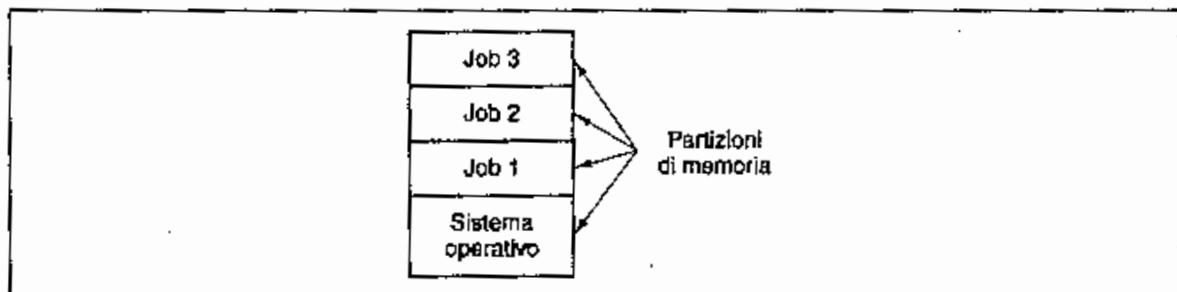


Figura 1.4 Un sistema multiprogrammato con tre job in memoria.

libro, i termini *procedura*, *subroutine* e *funzione* verranno utilizzati in modo equivalente) di cinque pagine", piuttosto che di comandi lunghi del tipo "ordina un migliaio di dati su nastro", il calcolatore è in grado di rendere disponibile un servizio interattivo molto veloce ad un certo numero di utenti e, forse, di lavorare contemporaneamente su un certo insieme di lavori organizzati in batch in background, nei momenti in cui la CPU sarebbe altrimenti inattiva. Sebbene il primo vero sistema timesharing, CTSS (Compatible Time Sharing System) sia stato sviluppato al MIT su un 7094 appositamente modificato (Corbató e altri, 1962) questi sistemi non divennero realmente popolari finché non si diffuse, nei calcolatori della terza generazione, l'hardware necessario, dedicato alla protezione.

Dopo il successo del sistema CTSS, MIT, Bell Labs e General Electric (che all'epoca era uno dei principali costruttori di calcolatori) decisero di intraprendere lo sviluppo di uno "strumento di calcolo", una macchina che fosse in grado di supportare centinaia di utenti in timesharing. Il loro modello era le reti di distribuzione elettrica: quando avete bisogno di elettricità dovete semplicemente infilare una spina nella presa a parete e, come ci si può aspettare, tutta l'elettricità di cui avete bisogno diventa immediatamente disponibile. I progettisti del sistema, noto con il nome di MULTICS (MULTIplexed Information and Computing Service, sistema informativo e di calcolo multiplexato), immaginavano una macchina molto grande che mettesse a disposizione capacità di calcolo per tutti gli abitanti di Boston: all'epoca, l'idea che macchine molto più potenti del loro GE-645 sarebbero state vendute a milioni, per un migliaio di dollari, solo 30 anni più tardi, era pura fantascienza, esattamente come accade oggi per l'idea dei treni transatlantici sottomarini.

MULTICS ebbe alterne fortune: era stato progettato per reggere centinaia di utenti su una macchina poco più potente di un PC 386 Intel, sebbene con maggiore capacità di ingresso/uscita: questo non è pazzesco come sembra, dal momento che le persone, a quei tempi, sapevano come scrivere programmi piccoli ed efficienti, capacità che, in seguito, è andata persa. Ci sono molte ragioni per cui MULTICS non conquistò il mondo, non ultima il fatto che fosse scritto in PL/1 e che il compilatore PL/1 fosse in ritardo di anni e funzionasse a malapena quando finalmente venne rilasciato. Inoltre, MULTICS fu un progetto troppo ambizioso per il suo tempo, tanto quanto la macchina analitica di Charles Babbage nel diciannovesimo secolo.

Riassumendo, MULTICS introdusse molte idee fondamentali per la letteratura sui calcolatori, ma trasformarlo in un prodotto vero e proprio ed in un successo commerciale era molto più complicato di quanto chiunque si potesse aspettare. I Bell Labs abbandonarono il progetto e la General Electric abbandonò completamente il settore dei calcolatori; comunque, il M.I.T. insistette ed alla fine ottenne che MULTICS funzionasse. MULTICS fu infine venduto come prodotto commerciale dalla ditta che comprò il ramo calcolatori della GE (Honeywell) e installato da circa 80 grandi aziende ed università nel mondo. Sebbene fossero pochi, gli utenti MULTICS furono tenacemente fedeli: General Motors, Ford e l'agenzia di sicurezza nazionale degli Stati Uniti (U.S. National Security Agency) per esempio, spensero i loro sistemi MULTICS alla fine degli anni '90, 30 anni dopo il suo rilascio.

Al momento, il concetto di un calcolatore che fornisce un servizio è sfumato, ma potrebbe riproporsi sotto la forma di giganteschi server Internet centralizzati, ai quali si collegano macchine utente relativamente stupide, che lasciano eseguire la maggior parte del lavoro ai grossi server. La motivazione di questo fenomeno è probabilmente data dal fatto che la maggior parte delle persone non vuole dover amministrare sistemi sempre più complessi e delicati, preferendo demandare il lavoro ai professionisti che lavorano per la ditta a cui appartiene il server. Il commercio elettronico si sta già evolvendo in questa direzione, con varie aziende che hanno collezioni di negozi virtuali su server multiprocessore, ai quali si connettono le macchine più semplici dei clienti, il tutto con uno spirito molto simile a quello del progetto MULTICS.

Nonostante il suo insuccesso commerciale, MULTICS ebbe un'enorme influenza sui sistemi successivi; questo sistema, descritto in (Corbató e altri, 1972; Corbató e Vyssotsky, 1965; Daley e Dennis, 1968; Organick, 1972; Saltzer, 1974), ha tuttora un sito Web attivo, www.multicians.org, con una grossa quantità di informazioni relative al sistema, ai suoi progettisti e utenti.

Un altro sviluppo importante avvenuto durante la terza generazione, fu la crescita fenomenale dei minicalcolatori, a partire dal PDP-1 della DEC, nel 1961. PDP-1 aveva solo 4K parole da 18 bit, ma a 120 000 dollari per macchina (meno del 5 per cento del prezzo di un 7094) veniva venduto come il pane. Per certe applicazioni non numeriche, era veloce quanto un 7094 e diede vita ad un'intera nuova industria; esso fu presto seguito da una serie di altri PDP (che, a differenza degli IBM, erano tutti incompatibili fra di loro) che culminò con il PDP-11.

Uno degli informatici dei Bell Labs, che aveva collaborato precedentemente al progetto di MULTICS, Ken Thompson, trovò in seguito un piccolo PDP-7 inutilizzato e si mise a scrivere una versione ridotta (singolo utente) di MULTICS. Questo lavoro si sviluppò e più tardi divenne il sistema operativo **UNIX®**, un sistema diffuso nel mondo accademico, nelle agenzie governative americane e in molte aziende.

La storia di UNIX è già stata raccontata altrove (per esempio, in Salus, 1994) e una parte di essa è riportata nel Capitolo 10. Per ora è sufficiente sapere che, essendo il codice sorgente ampiamente disponibile, varie organizzazioni hanno sviluppato proprie versioni (incompatibili), il che ha portato al caos. Le due versioni principali, **System V**, sviluppato dalla AT&T, e **BSD** (Berkeley Software Distribution), sviluppato dall'Università della California a Berkeley, sono anche esse dotate di piccole varianti. Per fare in modo che sia possibile scrivere programmi che possano essere eseguiti su qualsiasi sistema UNIX, IEEE ha sviluppato uno standard per UNIX, detto **POSIX**, che è attualmente supportato dalla maggior parte delle versioni di UNIX. POSIX definisce un'interfaccia minima di chiamate di sistema che i sistemi UNIX conformi allo standard devono supportare e che, in realtà, è attualmente supportata anche da qualche altro sistema operativo.

Detto per inciso, vale la pena menzionare che, nel 1987, l'autore di questo libro ha rilasciato a scopo didattico un piccolo clone di UNIX, **MINIX**, funzionalmente molto simile a UNIX e conforme a POSIX. È disponibile un libro che descrive le sue operazioni interne e che riporta in appendice il suo codice (Tanenbaum e Woodhull, 1997). MINIX è disponibile gratuitamente (compreso tutto il codice sorgente) all'indirizzo Internet www.cs.vu.nl/~ast/minix.html.

Il desiderio di scrivere una versione professionale gratuita (contrapposta a quella didattica) di MINIX, fece sì che uno studente finlandese, Linus Torvalds, scrivesse Linux. Questo sistema venne sviluppato su MINIX, di cui originariamente supportava varie caratteristiche (ad esempio, il file system); anche se da allora è stato esteso in molti modi, ancora adesso conserva una grossa quantità di strutture di fondo comuni a MINIX e a UNIX (sul quale si basava il primo) e la maggior parte di quello che verrà detto di UNIX in questo libro si applica quindi altrettanto bene a System V, BSD, MINIX, Linux e alle altre versioni e cloni di UNIX.

1.2.4 La quarta generazione (dal 1980 ad oggi): i Personal Computer

Con lo sviluppo dei circuiti integrati su larga scala (LSI, Large Scale Integration), chip contenenti migliaia di transistor su un centimetro quadrato di silicio, sorse l'era del personal computer. In termini di architettura, i personal computer (inizialmente chiamati microcalcolatori), non erano molto diversi dai minicalcolatori della classe dei PDP-11, ma

erano certamente diversi in termini di prezzo. Come il minicalcolatore rese possibile che un dipartimento di una ditta, o di un'università, avesse il proprio calcolatore, il microprocessore rese possibile che un singolo individuo avesse il proprio personal computer.

Nel 1974, quando Intel mise sul mercato l'8080, la prima CPU a 8 bit di tipo generale, essa volle un sistema operativo per questo processore, anche per poterlo collaudare e incaricò uno dei suoi consulenti, Gary Kildall, di scriverne uno. Kildall e un amico per prima cosa costruirono un controllore per il floppy disk a 8 pollici della Shugart Associates appena rilasciato, che installarono sull'8080, producendo quindi il primo microcalcolatore con disco; quindi, Kildall scrisse il sistema operativo basato su disco CP/M (Control Program for Microcomputer). Poiché Intel non credette che i microcalcolatori basati su disco potessero avere un futuro, quando Kildall chiese i diritti di CP/M, Intel glieli accordò. Kildall allora fondò una ditta, la Digital Research, per promuovere lo sviluppo e la vendita di CP/M.

Nel 1977, Digital Research riscrisse CP/M per renderlo adatto ad essere eseguito sulla maggior parte dei microcalcolatori che utilizzavano l'8080, Zilog Z80 e altre CPU. Molti applicazioni furono scritte per CP/M, permettendogli di dominare completamente il mondo dei microcalcolatori per circa 5 anni.

Nei primi anni '80, IBM progettò il PC IBM e cercò del software compatibile con esso; contattò un certo Bill Gates perché cedesse la licenza del suo interprete BASIC e gli chiese se fosse a conoscenza di un sistema operativo adatto al loro PC e Gates suggerì loro di contattare la Digital Research, la ditta che allora deteneva il dominio mondiale dei sistemi operativi. Prendendo quella che fu sicuramente la peggiore decisione aziendale a memoria d'uomo, Kildall rifiutò di incontrare IBM, mandando al suo posto un dipendente e, per peggiorare la situazione, il suo avvocato rifiutò persino di firmare un accordo per non divulgare informazioni sul nuovo PC, che non era ancora stato annunciato. Di conseguenza, IBM tornò da Gates chiedendogli di realizzare un sistema operativo per loro.

Quando IBM tornò da lui, Gates si accorse che una ditta locale produttrice di calcolatori, la Seattle Computer Products, aveva un sistema operativo adatto allo scopo, il DOS (Disk Operating System), quindi li contattò e fece loro un'offerta (a quanto si dice, di \$50,000), che prontamente accettarono. Gates offrì quindi il pacchetto DOS/BASIC a IBM, che lo accettò, ma richiese alcune modifiche, per realizzare le quali Gates assunse Tim Paterson, che aveva scritto il DOS, come impiegato di Microsoft, la neonata compagnia di Gates; quindi il sistema modificato venne ridenominato MS-DOS (MicroSoft Disk Operating System) e rapidamente cominciò a dominare il mercato dei PC IBM. Un fattore chiave di questo fenomeno fu la decisione, a posteriori estremamente saggia, di Gates di vendere MS-DOS alle aziende che producevano calcolatori, perché formasse un unico pacchetto con l'hardware, a differenza del tentativo iniziale di Kildall di vendere CP/M direttamente agli utenti finali, un sistema alla volta (almeno agli inizi).

Quando PC/AT di IBM venne rilasciato nel 1983 con la CPU Intel 80286, MS-DOS si era saldamente affermato e CP/M era ormai allo stremo. MS-DOS fu ampiamente utilizzato anche in seguito sui processori 80386 e 80486. Sebbene la versione iniziale di MS-DOS fosse veramente primitiva, le versioni successive inclusero caratteristiche più avanzate, molte di queste prese da UNIX. (Microsoft fu ben consapevole di UNIX, arrivando a venderne una versione per microcalcolatore chiamata XENIX, durante i primi anni dell'azienda.)

CP/M, MS-DOS e gli altri sistemi operativi per i primi microcalcolatori furono tutti basati sull'inserimento di comandi da tastiera da parte degli utenti, caratteristica che venne alla fine cambiata grazie alle ricerche fatte da Doug Engelbart allo Stanford Research Institute negli anni '60. Engelbart inventò la GUI (Graphical User Interface, o Interfaccia utente grafica), completa di finestre, icone, menu e mouse; queste idee vennero adottate dai ricercatori allo Xerox PARC e incorporate nelle macchine che costruiranno.

Un giorno, Steve Jobs, coinventore, nel suo garage, del calcolatore Apple, visitò PARC, vide una GUI e istantaneamente ne realizzò il valore potenziale, diversamente, com'è

noto, dalla direzione di Xerox (Smith e Alexander, 1988). Jobs si avventurò quindi nella costruzione di un Apple con una GUI. Questo progetto portò a Lisa, che si rivelò troppo costoso e fallì commercialmente. Il secondo tentativo di Job, il Macintosh di Apple, ebbe un enorme successo, non solo perché molto più economico di Lisa, ma anche perché fu **user friendly** (amichevole), era cioè destinato a utenti che non solo non conoscevano niente di calcolatori, ma che non avevano assolutamente intenzione di imparare alcunché in proposito.

Quando Microsoft decise di costruire il successore di MS-DOS, fu fortemente influenzata dal successo di Macintosh e produsse Windows, una sistema dotato di interfaccia grafica che, originariamente, veniva eseguito sopra MS-DOS (cioè fu più vicino a una shell che a un vero sistema operativo). Per circa 10 anni, dal 1985 al 1995, Windows fu solo un ambiente grafico al di sopra di MS-DOS, ma dal 1995 venne rilasciata una versione a sé stante di Windows, Windows 95, che incorporava molte caratteristiche di sistema operativo, usando il sottostante sistema MS-DOS solo per l'inizializzazione e per l'esecuzione di vecchi programmi MS-DOS. Nel 1998, venne rilasciato Windows 98, una versione leggermente modificata di questo sistema. Tuttavia, sia Windows 95 che Windows 98 contenevano ancora una grossa quantità di linguaggio assembly Intel a 16 bit.

Un altro sistema operativo Microsoft è Windows NT (NT sta per New Technology, cioè nuova tecnologia), che, per certi aspetti, è compatibile con Windows 95, ma, grattando la superficie, è una vera e propria riscrittura. È un sistema completamente a 32 bit, il cui progettista principale fu David Cutler, che fu anche uno dei progettisti del sistema operativo VAX VMS, per cui NT ne riporta alcune idee. Microsoft si aspettava che la prima versione di NT, essendo un sistema estremamente superiore, avrebbe eliminato MS-DOS e tutte le altre versioni di Windows, ma così non fu. Solo con la versione 4.0 alla fine Windows NT riuscì a prendere piede, in particolare presso le reti aziendali e all'inizio del 1999, la versione 5 di Windows NT venne ribattezzata Windows 2000. Nelle intenzioni, questo sistema avrebbe dovuto rappresentare il successore sia di Windows 98 che di Windows NT 4.0, ma non andò così e Microsoft rilasciò un'altra versione di Windows 98, Windows Me (**Millennium edition**, edizione del millennio).

L'altro concorrente principale nel mondo dei personal computer è UNIX (e i suoi vari derivati), che domina sulle workstation e sugli altri calcolatori di fascia alta, come i server di rete, ed è particolarmente popolare sulle macchine dotate di chip RISC ad alte prestazioni. Sui calcolatori con processore Pentium, Linux si sta affermando, fra gli studenti e sempre più anche fra le aziende, come un'alternativa a Windows (per tutto il libro, useremo il termine "Pentium" per intendere i processori Pentium I, II, III e 4).

Sebbene molti utenti UNIX, in particolare programmati esperti, preferiscono un'interfaccia a comandi a una GUI, quasi tutti i sistemi UNIX forniscono X Windows, un sistema a finestre prodotto al M.I.T. che offre la gestione di base delle finestre, permettendo all'utente di crearle, cancellarle, muoverle, ridimensionarle usando un mouse. Spesso è disponibile anche una GUI completa, come Motif, da eseguire sopra il sistema X Windows, per dare a UNIX, se lo si desidera, un aspetto (look and feel) simile a quello di Macintosh o di Microsoft Windows.

Uno sviluppo interessante, che iniziò a prendere piede durante la metà degli anni '80, è la crescita di reti di personal computer su cui girano sistemi operativi di rete (network operating system) e sistemi operativi distribuiti (distributed operating system: si veda Tanenbaum e Van Steen, 2002). In un sistema operativo di rete, gli utenti sono consapevoli dell'esistenza di diversi calcolatori, possono accedere a macchine remote e copiare file da una macchina all'altra: ogni macchina ha il proprio sistema operativo locale ed un proprio utente (o propri utenti).

I sistemi operativi di rete non sono diversi, nella sostanza, dai sistemi operativi a singolo processore: ovviamente, necessitano di un controllore per l'interfaccia di rete e del

software a basso livello per il suo funzionamento, oltre che di programmi per eseguire il login remoto e l'accesso a file in remoto, ma queste aggiunte non cambiano la struttura essenziale del sistema operativo.

Viceversa, un sistema operativo distribuito appare ai suoi utenti come un sistema monoprocesso tradizionale, anche se è effettivamente composto da diversi processori. In un vero sistema distribuito gli utenti non dovrebbero sapere dove vengono fatti girare i loro programmi o dove siano allocati i loro file; tutto ciò dovrebbe essere gestito dal sistema operativo in maniera automatica ed efficiente.

I veri sistemi operativi distribuiti richiedono più della semplice aggiunta di un po' di codice ad un sistema operativo uniprocesso, poiché sistemi distribuiti e centralizzati differiscono per caratteristiche essenziali: i sistemi distribuiti, per esempio, spesso permettono ai programmi di girare contemporaneamente su diversi processori, quindi richiedono algoritmi di pianificazione dell'utilizzo del processore più complessi per ottimizzare l'esecuzione parallela.

I ritardi delle comunicazioni sulla rete fanno sì che spesso questi (ed altri) algoritmi debbano girare utilizzando informazioni incomplete, non aggiornate o perfino scorrette. Una tale situazione è radicalmente diversa da quella di un sistema a singolo processore, dove il sistema operativo ha conoscenza completa dello stato del sistema.

1.2.5 L'ontogenesi riassume la filogenesi

Dopo la pubblicazione del libro *L'origine delle specie* di Charles Darwin, lo zoologo tedesco Ernst Haeckel dichiarò che "L'ontogenesi riassume la filogenesi", intendendo con questo che lo sviluppo di un embrione (ontogenesi) ripete, riassumendola, l'evoluzione della specie (filogenesi). In altre parole, dopo essere stato fecondato, un ovulo umano passa attraverso stadi intermedi in cui è un pesce, un maiale e così via, fino a diventare un neonato. I biologi moderni considerano questa affermazione una semplificazione grossolana, ma c'è un nocciolo di verità in essa.

Qualcosa di analogo è accaduto all'industria dei calcolatori, dove ogni nuova specie (mainframe, minicalcolatore, personal computer, calcolatore embedded, smart card, eccetera) sembra passare per le fasi di evoluzione che i suoi antenati hanno già attraversato. I primi mainframe venivano completamente programmati in linguaggio assembler, con il quale venivano scritti persino programmi complessi, come compilatori e sistemi operativi. Al tempo della comparsa dei minicalcolatori, FORTRAN, COBOL e altri linguaggi ad alto livello erano diventati comuni sui mainframe, nonostante che i nuovi minicalcolatori, per mancanza di memoria, venissero ancora programmati in assembler. Quando furono inventati i microcalcolatori (i primi personal computer), anche questi venivano programmati in assembler, anche se, all'epoca, i minicalcolatori venivano programmati usando linguaggi ad alto livello. Anche la programmazione sui calcolatori palmari è cominciata con codice assembler, per arrivare velocemente all'uso dei linguaggi ad alto livello (principalmente perché il lavoro di sviluppo veniva fatto su macchine più grosse); lo stesso dicasi per le smart card (carte intelligenti).

Diamo ora un'occhiata ai sistemi operativi. I primi mainframe, inizialmente, non avevano hardware di protezione né supporto per la multiprogrammazione, così su di essi giravano sistemi operativi semplici, che gestivano un solo programma alla volta, caricato a mano. Più tardi, essi furono estesi con caratteristiche hardware e di sistema operativo per eseguire più programmi alla volta, ed infine con vere e proprie possibilità d'uso in timesharing.

Quando comparvero i primi minicalcolatori, anche essi non avevano hardware di protezione ed eseguivano un programma per volta, caricato manualmente, nonostante che

la multiprogrammazione fosse già ben consolidata, all'epoca, sui mainframe; poi, gradatamente, essi acquisirono hardware di protezione e la capacità di eseguire due o più programmi alla volta. Allo stesso modo, il primo microcalcolatore era inizialmente in grado di eseguire un programma per volta e, successivamente, acquisì la capacità di lavorare in multiprogrammazione, strada seguita anche dai calcolatori palmari e dalle smart card.

I dischi apparvero inizialmente sui principali mainframe, quindi sui minicalcolatori, poi sui microcalcolatori e così via; fino ad ora, le smart card non hanno un disco rigido, ma, con l'avvento delle memorie flash ROM, avranno presto un dispositivo equivalente. Quando i dischi apparvero, spuntarono i primi sistemi operativi. Sul CDC 6600, senza dubbio il più potente mainframe del mondo per quasi tutti gli anni '60, il file system era composto dagli utenti, che avevano la possibilità di creare un file e dichiararlo quindi permanente, intendendo con questo che il file rimaneva sul disco anche dopo che il programma che l'aveva creato era terminato. Per accedere a questo file in seguito, un programma doveva aprirlo con un comando speciale, utilizzando una parola chiave, fornita nel momento in cui il file era stato reso permanente: di fatto, c'era una sola directory condivisa da tutti gli utenti, ai quali spettava il compito di evitare conflitti di nome fra i file. Anche i primi file system dei minicalcolatori e dei microcalcolatori avevano una sola directory condivisa fra tutti gli utenti.

La memoria virtuale, che consente di eseguire programmi di dimensione maggiore della memoria fisica, ha avuto uno sviluppo simile. Comparve prima sui mainframe, poi sui minicalcolatori, sui microcalcolatori e, gradatamente, su sistemi sempre più piccoli. Le reti di calcolatori hanno una storia simile.

In tutti i casi, lo sviluppo del software è stato comandato dalla tecnologia. I primi microcalcolatori, per esempio, avevano qualcosa come 4 KB di memoria e nessun hardware di protezione e i linguaggi ad alto livello e la multiprogrammazione erano troppo difficili da gestire su sistemi così piccoli. Come i microcalcolatori crebbero nei personal computer moderni, essi acquisirono prima l'hardware, quindi il software necessario per gestire caratteristiche più avanzate: l'evoluzione probabilmente continuerà in questo modo per gli anni a venire. Altri settori sembrerebbero avere questo sviluppo ciclico, simile ad una reincarnazione, ma nell'industria dei calcolatori, la rotazione è più veloce.

1.3 Lo zoo dei sistemi operativi

La storia di quest'evoluzione ci ha lasciato con un'ampia varietà di sistemi operativi, alcuni dei quali poco conosciuti, sette dei quali vengono trattati brevemente in questa sezione. Torneremo su alcuni di questi diversi tipi di sistemi in seguito.

1.3.1 Sistemi operativi per i mainframe

Al livello più alto troviamo i sistemi operativi per i mainframe, quei calcolatori della dimensione di una stanza tuttora presenti nei centri di elaborazione dati delle principali aziende. Essi si distinguono dai personal computer per la loro capacità di ingresso / uscita (I/O): un mainframe con 1000 dischi e migliaia di gigabyte di dati non è insolito, mentre un personal computer con le stesse caratteristiche sarebbe veramente strano. I mainframe stanno ritornando anche come Web server di fascia alta, come server per siti di commercio elettronico su larga scala e come server per la gestione di transazioni commerciali.

I sistemi operativi per i mainframe sono fortemente orientati all'elaborazione di molti job per volta, la maggior parte dei quali necessita di una quantità enorme di ingre-

so/uscita. Generalmente, offrono tre tipi di servizi: elaborazione batch, elaborazione di transazioni e elaborazione con condivisione rispetto al tempo. Un sistema batch elabora job senza alcuna interazione con l'utente. L'elaborazione delle pratiche di liquidazione in una azienda assicurativa o il riepilogo delle vendite per una catena di negozi sono elaborazioni tipicamente di tipo batch. I sistemi che elaborano transazioni manipolano grosse quantità di piccole richieste come, ad esempio, l'elaborazione degli assegni in una banca o le prenotazioni aeree, dove ogni unità di lavoro è di piccola dimensione, ma il sistema ne deve manipolare centinaia o migliaia al secondo. I sistemi a tempo condiviso permettono a molti utenti remoti di eseguire contemporaneamente i loro job sul calcolatore, come, ad esempio, le interrogazioni in una grossa base di dati. Queste funzioni sono strettamente correlate e i sistemi operativi dei mainframe spesso le supportano tutte; un esempio di sistema operativo per mainframe è OS/390, un discendente di OS/360.

1.3.2 Sistemi operativi per i server

Scendendo di un livello troviamo i sistemi operativi per i server, che possono essere personal computer molto grandi, postazioni di lavoro o persino mainframe, e servono più utenti in rete contemporaneamente, permettendo loro di condividere risorse hardware e software. I server possono fornire servizi di stampa, servizi relativi ai file o servizi Web. I fornitori di accesso a Internet (Internet provider) utilizzano molti server per gestire i servizi per i loro clienti, e i siti Web utilizzano i server per memorizzare le pagine Web e gestire le richieste in arrivo. I sistemi operativi comunemente utilizzati per i server sono UNIX e Windows 2000, ma anche Linux sta guadagnando terreno in questo settore.

1.3.3 Sistemi operativi multiprocessore

Un modo comune sempre più utilizzato per aumentare la potenza di calcolo è unire più CPU in un unico sistema e, a seconda di come sono precisamente collegate e di cosa viene condiviso, questi sistemi vengono chiamati calcolatori paralleli, multicalcolatori o multiprocessori. Questi sistemi necessitano di sistemi operativi particolari, che spesso sono varianti dei sistemi operativi per i server, con caratteristiche particolari per la comunicazione e la connessione.

1.3.4 Sistemi operativi per i personal computer

La categoria successiva è quella dei sistemi operativi per i personal computer, il cui compito è quello di fornire una buona interfaccia all'utente singolo e che sono ampiamente utilizzati per l'elaborazione di testi, l'uso di fogli di calcolo e l'accesso a Internet. Esempi tipici sono Windows 98, Windows 2000, il sistema operativo per Macintosh (MacOS) e Linux. I sistemi operativi per i personal computer sono così ampiamente conosciuti che, probabilmente, necessiterebbero di poche introduzioni. Effettivamente, molte persone non sono neppure a conoscenza dell'esistenza di altri sistemi.

1.3.5 Sistemi operativi real-time

Un altro tipo di sistema operativo è il sistema operativo per i sistemi real-time (tempo reale), che sono caratterizzati dall'avere il tempo come parametro chiave. Per esempio, nei sistemi di controllo dei processi industriali, i calcolatori real-time devono raccogliere i dati relativi al processo di produzione, e usarli per controllare i macchinari nello stabilimento.

Vi sono spesso scadenze strette da rispettare: ad esempio, se un'auto si sta muovendo su una catena di montaggio, si devono eseguire determinate azioni ad istanti di tempo precisi, e se un robot saldatore salda troppo presto o troppo tardi, l'auto verrà rovinata. Se l'azione *dove* assolutamente essere eseguita in un certo momento (o in un determinato intervallo di tempo), abbiamo un **sistema real-time stretto** (hard real-time system).

Un altro tipo di sistema real-time è un **sistema real-time lasco** (soft real-time system), nel quale è accettabile non rispettare talvolta una scadenza. I sistemi digitali audio o i sistemi multimediali appartengono a questa categoria. VxWorks e QNX sono sistemi operativi in real-time molto conosciuti.

1.3.6 Sistemi operativi embedded

Continuando a scendere verso sistemi sempre più piccoli, arriviamo ai calcolatori palmari e ai sistemi embedded. Un calcolatore palmare o PDA (Personal Digital Assistant) è un piccolo calcolatore che sta nella tasca di una camicia ed esegue alcune funzioni come quelle di agenda degli indirizzi e promemoria degli appuntamenti. I sistemi embedded vengono eseguiti su calcolatori che controllano dispositivi che non sono generalmente nati come calcolatori, ad esempio, televisori, forni a microonde, telefoni cellulari. Questi hanno spesso delle caratteristiche dei sistemi in tempo reale, ma hanno anche dimensione, memoria e limitazioni di alimentazione che li rendono particolari. Esempi di questi sistemi operativi sono PalmOS e Windows CE (CE sta per Consumer Electronics).

1.3.7 Sistemi operativi per le smart card

I sistemi operativi più piccoli vengono eseguiti sulle smart card, che sono dispositivi della dimensione di una carta di credito che contengono un chip di CPU. Hanno stretti vincoli sull'alimentazione e sulla memoria; alcuni di essi possono eseguire una sola funzione, come il pagamento elettronico, mentre altri ne supportano più d'una sulla stessa smart card; spesso sono sistemi proprietari.

Alcune smart card sono orientate a Java, cioè la ROM sulle smart card contiene un interprete per la Java Virtual Machine (JVM). Le applet Java (piccoli programmi) vengono scaricate sulla carta e sono interpretate dall'interprete JVM. Alcune di queste carte possono gestire più applet Java contemporaneamente, portando alla multiprogrammazione e alla necessità di pianificare l'esecuzione; ma anche la gestione e la protezione delle risorse diventano un problema se vengono eseguiti due o più applet contemporaneamente. Questi problemi devono essere gestiti dal sistema operativo presente sulla card, che, di solito, è estremamente primitivo.

1.4 Rassegna di hardware per calcolatori

Un sistema operativo è profondamente legato all'hardware del calcolatore sul quale viene eseguito, del quale estende l'insieme di istruzioni e gestisce le risorse. Per lavorare, deve conoscere molto circa l'hardware o, almeno, molto di come l'hardware appare al programmatore.

Concettualmente, un semplice personal computer può essere modellato come in Figura 1.5: CPU, memoria e dispositivi di ingresso/uscita sono connessi da un bus di sistema attraverso il quale comunicano fra di loro. I personal computer moderni hanno una struttura molto più complicata, con più di un bus, che vedremo più avanti; per ora, questo

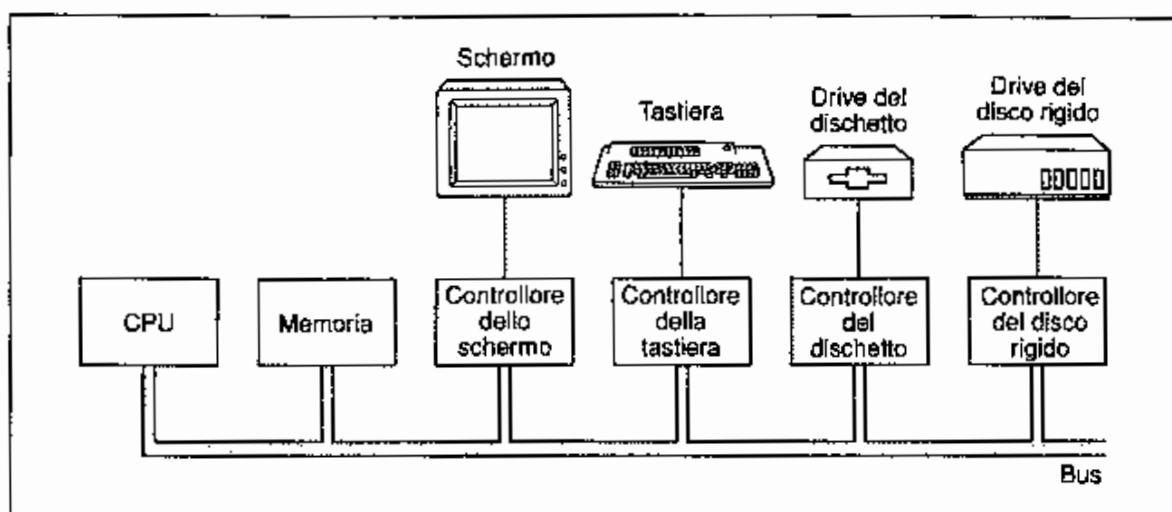


Figura 1.5 Alcune componenti di un semplice personal computer

modello sarà sufficiente. Nelle sezioni seguenti, passeremo brevemente in rassegna queste componenti, ed esamineremo alcuni dei problemi hardware che sono di competenza dei progettisti di sistemi operativi.

1.4.1 Processori

Il "cervello" di un calcolatore è la CPU, che recupera le istruzioni dalla memoria e le esegue. Il ciclo di azioni base eseguito da ogni CPU consiste nel recuperare la prima istruzione dalla memoria, decodificarla per determinarne tipo e operandi, ed eseguirla e quindi recuperare, decodificare, eseguire le istruzioni seguenti. In questo modo vengono completati i programmi.

Ogni CPU è in grado di eseguire un insieme di istruzioni specifico; di conseguenza, un processore Pentium non può eseguire un programma scritto per un processore SPARC e viceversa. Poiché l'accesso alla memoria per recuperare un'istruzione o un dato richiede più tempo che non eseguire un'istruzione, tutte le CPU contengono, al loro interno, alcuni registri per mantenervi variabili chiave e risultati temporanei. Normalmente, l'insieme di istruzioni ne contiene quindi alcune per caricare una parola dalla memoria in un registro e per depositare una parola da un registro alla memoria. Altre istruzioni combinano due operandi da registro, memoria o da entrambi, ottenendo un risultato, ad esempio sommare due parole e mettere il risultato in un registro o in memoria.

In aggiunta ai registri generici usati per contenere le variabili e i risultati temporanei, la maggior parte dei calcolatori ha parecchi registri speciali disponibili al programmatore. Uno di questi è il **program counter**, o **contatore di programma**, che contiene l'indirizzo di memoria della prossima istruzione da andare a prendere (fetch); dopo che l'istruzione è stata recuperata, il program counter viene aggiornato per puntare all'istruzione successiva.

Un altro registro è lo **stack pointer** o **puntatore alla pila**, che punta alla cima dello stack corrente in memoria; tale stack o pila contiene una struttura (frame) per ogni procedura che è stata iniziata ma non è ancora terminata, ognuna delle quali contiene i parametri d'ingresso, e le variabili locali e temporanee che non vengono mantenuti nei registri.

Ancora, un altro registro è il **PSW** (**Program Status Word** o **parola di stato del programma**), che contiene i bit di codice di condizione, che vengono inizializzati da istru-

zioni di confronto, la priorità della CPU, la modalità (kernel o utente) e altri svariati bit di controllo. I programmi utente possono normalmente leggere l'intero PSW, ma tipicamente possono scrivere solo alcuni suoi campi; il PSW gioca un ruolo importante per le chiamate di sistema e l'ingresso/uscita.

Il sistema operativo deve essere consapevole di tutti i registri: quando ripartisce la CPU rispetto al tempo, blocca spesso il programma in esecuzione per farne (ri)partire un altro e, ogni volta, deve salvare tutti i registri in modo che possano essere ripristinati quando, successivamente, il programma sarà di nuovo in esecuzione.

Per migliorare le prestazioni, i progettisti di CPU hanno da tempo abbandonato il modello semplificato che prevede il recupero, la decodifica e l'esecuzione di un'istruzione per volta. Molte CPU moderne hanno dispositivi per eseguire più istruzioni contemporaneamente; per esempio, una CPU potrebbe avere unità separate per le tre fasi di recupero, decodifica e esecuzione, così che, mentre è in esecuzione l'istruzione n , potrebbe essere in grado di decodificare l'istruzione $n+1$ e recuperare l'istruzione $n+2$. Un'organizzazione di questo tipo è detta **pipeline**; in Figura 1.6(a) è illustrata una pipeline a tre stadi, ma sono comuni anche pipeline più lunghe. Nel progetto di molte pipeline, una volta che un'istruzione è stata recuperata e posta nella pipeline, deve essere eseguita, anche se l'istruzione precedente era una condizione, e si è selezionato il ramo opposto. Le pipeline procurano grossi mal di testa a chi scrive compilatori e sistemi operativi, perché mostrano loro la complessità della macchina sottostante.

Persino più avanzata del modello a pipeline è la CPU **superscalare** mostrata in Figura 1.6(b): in questo modello sono presenti unità multiple di esecuzione, per esempio una per l'aritmetica intera, una per l'aritmetica a virgola mobile, e una per le operazioni booleane. Vengono recuperate due o più istruzioni per volta, decodificate e riversate in un buffer (area di memoria temporanea), fino a quando potranno essere eseguite. Non appena un'unità di esecuzione si libera, controlla se nel buffer è presente un'istruzione che è in grado di gestire e, se è così, la rimuove dal buffer e la esegue. Una conseguenza di questo progetto è che le istruzioni di programma spesso non vengono eseguite nell'ordine in cui sono scritte; per la maggior parte, è compito dell'hardware assicurare che il risultato prodotto sia lo stesso di quello che produrrebbe un'implementazione sequenziale, ma ciò impone una fastidiosa aggiunta di complessità al sistema operativo, come vedremo.

La maggior parte delle CPU, eccetto quelle molto semplici utilizzate nei sistemi embedded, hanno due modalità, modalità kernel e modalità utente, come detto in precedenza. Generalmente un bit nel registro PSW controlla la modalità, e quando è in esecuzione in modalità kernel, la CPU può eseguire qualsiasi istruzione nel suo insieme di istruzioni ed usare ogni caratteristica dell'hardware. Il sistema operativo viene eseguito in modalità kernel, il che gli permettendo l'accesso completo all'hardware.

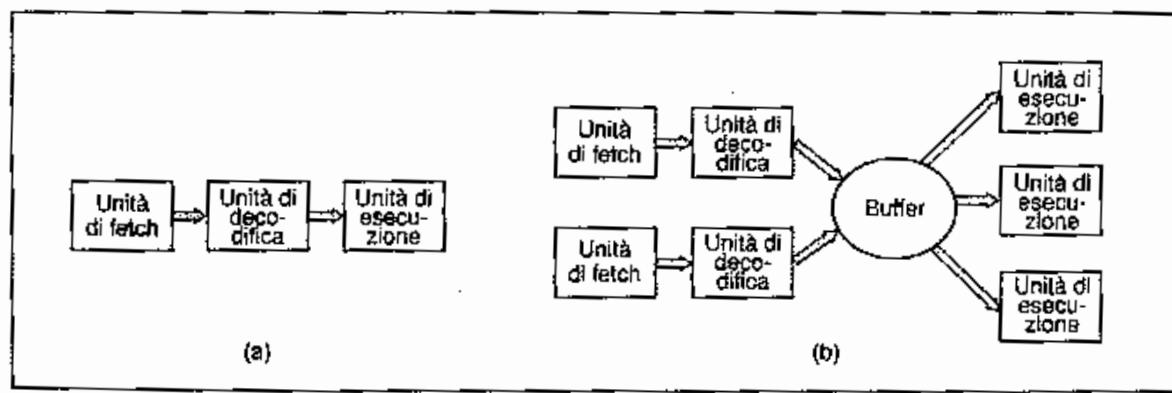


Figura 1.6 (a) Una pipeline a tre stadi. (b) Una CPU superscalare.

Al contrario, i programmi utente vengono eseguiti in modalità utente, che permette di eseguire solo un sottoinsieme di istruzioni e di accedere a un sottoinsieme di caratteristiche hardware. Di solito, nella modalità utente sono vietate tutte le istruzioni che coinvolgono l'ingresso/uscita e la protezione della memoria; ovviamente, è anche vietato impostare il bit di modalità del registro PSW.

Per ottenere servizi dal sistema operativo, un programma deve eseguire una chiamata di sistema o system call, che esegue una trap al kernel e invoca il sistema operativo: l'istruzione TRAP cambia da modalità utente a modalità kernel e avvia il sistema operativo. Quando il lavoro è stato completato, il controllo ritorna al programma utente all'istruzione successiva alla chiamata di sistema; i dettagli del trattamento delle chiamate di sistema saranno esposti più avanti in questo capitolo.

Come nota di tipografia, useremo un carattere speciale per indicare le chiamate di sistema nel testo, come questo: `read`.

È bene notare che gli elaboratori hanno altre istruzioni di tipo trap oltre a quella per eseguire chiamate di sistema: la maggior parte delle trap sono causate dall'hardware per avvisare di una situazione eccezionale, come può essere un tentativo di divisione per 0 o un underflow nell'aritmetica a virgola mobile. In ogni caso, il sistema operativo ottiene il controllo, e deve decidere cosa fare; spesso il programma deve terminare con un errore; altre volte, l'errore può essere ignorato (un numero andato in underflow può essere posto a 0). Per finire, se il programma ha annunciato in precedenza che vuole gestire certi tipi di condizioni, il controllo può essergli ripassato per permettergli di occuparsi del problema.

1.4.2 Memoria

La seconda componente importante in ogni elaboratore è la memoria; la memoria ideale dovrebbe essere estremamente veloce (più veloce dell'esecuzione di un'istruzione, in modo che la CPU non sia rallentata dalla memoria), abbondante, ed a bassissimo costo. Nessuna tecnologia attuale soddisfa tutti questi obiettivi, così si è adottato un approccio diverso: la memoria è costruita come una gerarchia di strati, come mostrato in Figura 1.7.

Lo strato più alto si compone dei registri interni alla CPU, essi sono costruiti con la stessa tecnologia della CPU e sono quindi veloci quanto questa; di conseguenza, non c'è nessun ritardo accedendo ad essi. La capacità di memorizzazione disponibile è di 32×32 bit su una CPU a 32 bit, e 64×64 bit su una CPU a 64 bit: meno di 1 KB in entrambi i casi. I programmi devono gestire i registri (per esempio decidere cosa memorizzarci) via software.

Subito dopo, viene la memoria cache, che è, per lo più, controllata dall'hardware. La memoria principale è divisa in linee di cache, tipicamente di 64 byte, con indirizzo da 0 a 63 nella linea di cache 0, da 64 a 127 nella linea di cache 1 e così via. Le linee di cache usate più frequentemente sono mantenute in una cache ad alta velocità, situata all'interno o comunque molto vicino alla CPU. Quando il programma ha bisogno di leggere una parola di memoria, l'hardware della cache controlla se la linea richiesta è presente nella cache. Se è così, si ha un cache hit (letteralmente, cache colpita), la richiesta è soddisfatta dalla cache e nessuna richiesta viene mandata alla memoria principale attraverso il bus. I cache hit, normalmente, durano due cicli di clock, mentre i cache miss (letteralmente, cache mancata) devono rivolgersi alla memoria, con una considerevole penalizzazione in termini di tempo. La memoria cache ha dimensione limitata a causa del suo alto costo, ma alcune macchine hanno due o persino tre livelli di cache, ognuno più lento e più grande di quello che lo precede.

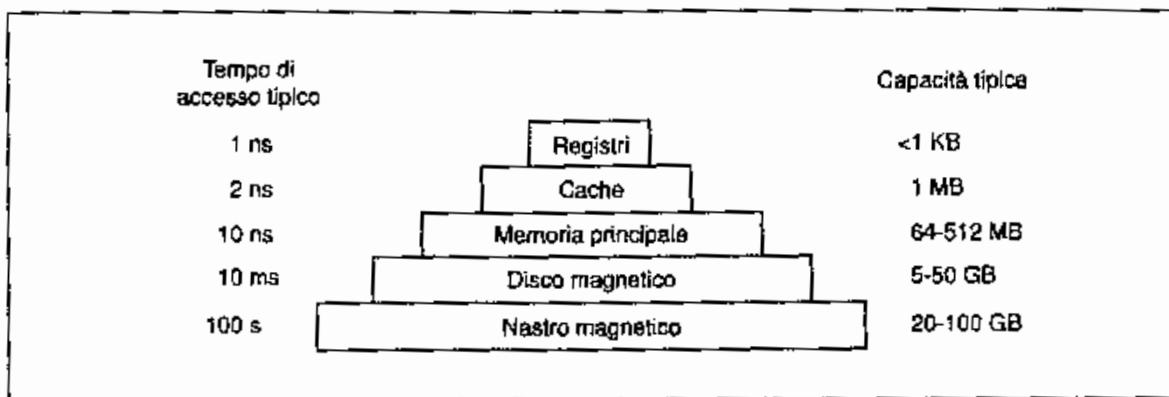


Figura 1.7 Una gerarchia di memoria tipica. I numeri sono approssimazioni grossolane.

Di seguito viene la memoria principale, il cavallo da tiro del sistema delle memorie. La memoria principale è spesso chiamata **RAM** (Random Access Memory, memoria ad accesso casuale), ma un tempo era detta **memoria a nucleo** (core memory), perché la memoria principale degli elaboratori negli anni '50 e '60 era realizzata mediante sottili nuclei di ferrite magnetizzabile. Attualmente, le memorie hanno dimensione di decine o centinaia di megabyte che crescono rapidamente; tutte le richieste della CPU che non possono essere soddisfatte dalla memoria cache vanno alla memoria principale.

Il prossimo nella gerarchia è il disco magnetico o disco fisso (hard disk). La memorizzazione su disco è due ordini di grandezza più economica della RAM per bit, e spesso anche due ordini di grandezza più ampia. L'unico problema è che il tempo per l'accesso casuale ai dati presenti su un disco è quasi tre ordini di grandezza più lungo; questa bassa velocità è dovuta al fatto che il disco è un dispositivo meccanico, come mostrato in Figura 1.8.

Un disco si compone di uno o più piatti metallici che ruotano a 5400, 7200 o 10800 rpm (round per minute, giri al minuto). Un braccio meccanico gira su un perno al di sopra dei piatti dall'angolo, in modo simile al braccio della puntina di un vecchio giradischi a 33 giri per dischi in vinile. Le informazioni sono scritte sul disco come una serie di cerchi concentrici. In ogni posizione del braccio, ciascuna testina legge una regione a forma di anello detta traccia (track); tutte assieme, le tracce per una data posizione del braccio formano un cilindro.

Ogni traccia è divisa in un certo numero di settori, generalmente di 512 byte ognuno; sui dischi moderni, i cilindri esterni contengono più settori di quelli interni. Lo spostamento del braccio da un cilindro a quello contiguo impiega circa 1 ms; per spostarlo in un cilindro a caso tipicamente occorrono da 5 ms a 10 ms, in base all'unità. Una volta che il braccio è posizionato sulla traccia corretta, l'unità deve aspettare che il settore passi ruotando sotto la testina, un ritardo addizionale da 5 ms a 10 ms, in base al numero di giri al minuto dell'unità, ed una volta che il settore è sotto alla testina, la lettura o la scrittura avvengono con un ritmo di 5 MB/s su dischi di fascia bassa, ma fino a 160 MB/s su quelli più veloci.

L'ultimo strato nella gerarchia di memoria è il nastro magnetico. Questo supporto è spesso usato come backup (copia di riserva) per il contenuto del disco, e per mantenere grosse moli di dati. Per accedere a un nastro, esso deve prima essere messo in un lettore di nastri, o da una persona o da un robot (la manipolazione automatica dei nastri è comune in installazioni con basi di dati molto grandi). Dopodiché il nastro si deve srotolare fino ad arrivare al blocco richiesto: tutto sommato, ciò può richiedere minuti. Il grande vantaggio del nastro è che ha un costo per bit oltremodo basso e che è rimo-

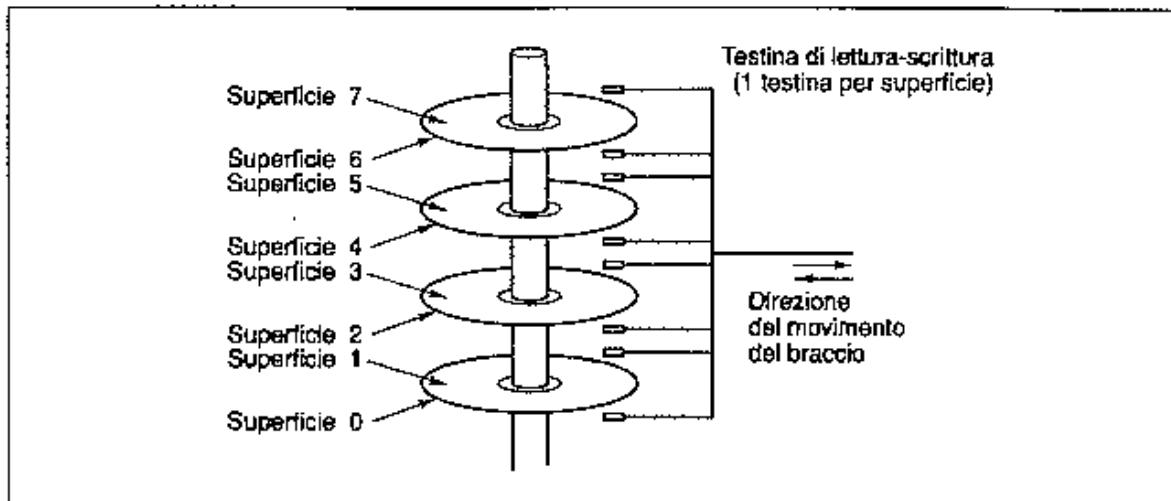


Figura 1.8 Struttura di un drive di disco.

sotikaKes

vibile, fatto importante per nastri di backup che devono essere conservati *off-site*, cioè lontano dal computer, per sopravvivere ad incendi, inondazioni, terremoti, ecc.

La gerarchia di memoria che abbiamo discusso è comune, ma alcune installazioni non hanno tutti gli strati, o ne hanno altri un po' diversi (come i dischi ottici). Tuttavia, in ognuna di esse, scendendo nella gerarchia, il tempo di accesso casuale cresce velocemente, e il costo per bit diminuisce enormemente; di conseguenza, è probabile che le gerarchie di memoria continueranno ad esistere negli anni a venire.

In aggiunta ai tipi di memoria discussi sopra, molti elaboratori hanno una piccola porzione di memoria ad accesso casuale non volatile; a differenza della RAM, la memoria non volatile non perde il suo contenuto nel momento in cui viene tolta la corrente. La ROM (Read Only Memory, memoria di sola lettura) è programmata in fabbrica e non può essere modificata in seguito; è veloce ed economica. Su qualche elaboratore, il programma di avvio (bootstrap loader) usato per avviare l'elaboratore stesso è contenuto in ROM; inoltre, alcune schede di ingresso/uscita contengono una ROM per la gestione del controllo del dispositivo a basso livello.

Anche le EEPROM (Electrically Erasable ROM, ROM cancellabili elettricamente) e le flash RAM non sono volatili, ma al contrario delle ROM possono essere cancellate e riscritte. Tuttavia, per scrivere su di esse occorrono tempi di qualche ordine di grandezza più lunghi di quelli per scrivere sulle RAM, perciò sono usate nello stesso modo delle ROM, con l'unica caratteristica aggiuntiva che è possibile correggere bachi nei programmi che contengono, riscrivendoli sul campo.

Un altro tipo di memoria è la memoria CMOS, che è volatile ed è utilizzata da molti elaboratori per memorizzare ora e data corrente. La memoria CMOS e il circuito dell'orologio che incrementa il tempo in essa sono alimentati da una piccola batteria, in modo che l'ora sia aggiornata correttamente anche quando l'elaboratore è scollegato dalla rete elettrica; questa memoria può anche contenere i parametri di configurazione, come il disco dal quale eseguire il boot. La memoria CMOS è utilizzata perché consuma così poca corrente che la batteria originale installata in fabbrica dura spesso diversi anni. Tuttavia, quando inizia a scaricarsi, può sembrare che l'elaboratore abbia il morbo di Alzheimer, dimenticandosi cose che ha saputo per anni, come il disco dal quale eseguire il boot.

Focalizziamo l'attenzione sulla memoria per un istante: spesso si desidera mantenere più programmi in memoria nello stesso istante; se un programma è bloccato in attesa che

si completi una lettura da disco, un altro programma può usare la CPU, migliorando la percentuale di utilizzo della CPU. Comunque, per tenere due o più programmi contemporaneamente nella memoria principale occorre risolvere due problemi:

1. Come proteggere i programmi l'uno dall'altro, e il kernel dai programmi.
2. Come gestire la rilocazione.

Sono possibili molte soluzioni, che, tuttavia, richiedono tutte di attrezzare la CPU con hardware speciale.

Il primo problema è ovvio, ma il secondo è un po' più sottile. Quando un programma è compilato e linkato, il compilatore e il linker non sanno dove verrà caricato, nella memoria fisica, quando sarà eseguito; per questa ragione, normalmente ipotizzano che inizierà all'indirizzo 0, ove metteranno la prima istruzione. Si supponga che la prima istruzione recuperi una parola dall'indirizzo di memoria 10000, e che tutto il programma e i dati siano caricati a partire dall'indirizzo 50000; quando la prima istruzione viene eseguita, fallirà perché farà riferimento alla parola all'indirizzo 10000 invece che alla parola all'indirizzo 60000. Per risolvere questo problema, si ha bisogno o di rilocare il programma quando viene caricato, trovando tutti gli indirizzi e modificandoli (che è fattibile, ma costoso), oppure di fare una rilocazione al volo durante l'esecuzione.

La soluzione più semplice è mostrata in Figura 1.9(a). Nella figura vediamo un elaboratore dotato di due registri speciali, il **registro base** e il **registro limite** (si noti che, nel libro, i numeri che iniziano con 0x sono esadecimali – secondo la convenzione del linguaggio C; in modo simile, i numeri che iniziano con uno zero sono ottali). Quando un programma viene eseguito, il registro base è impostato al punto di inizio del suo codice,

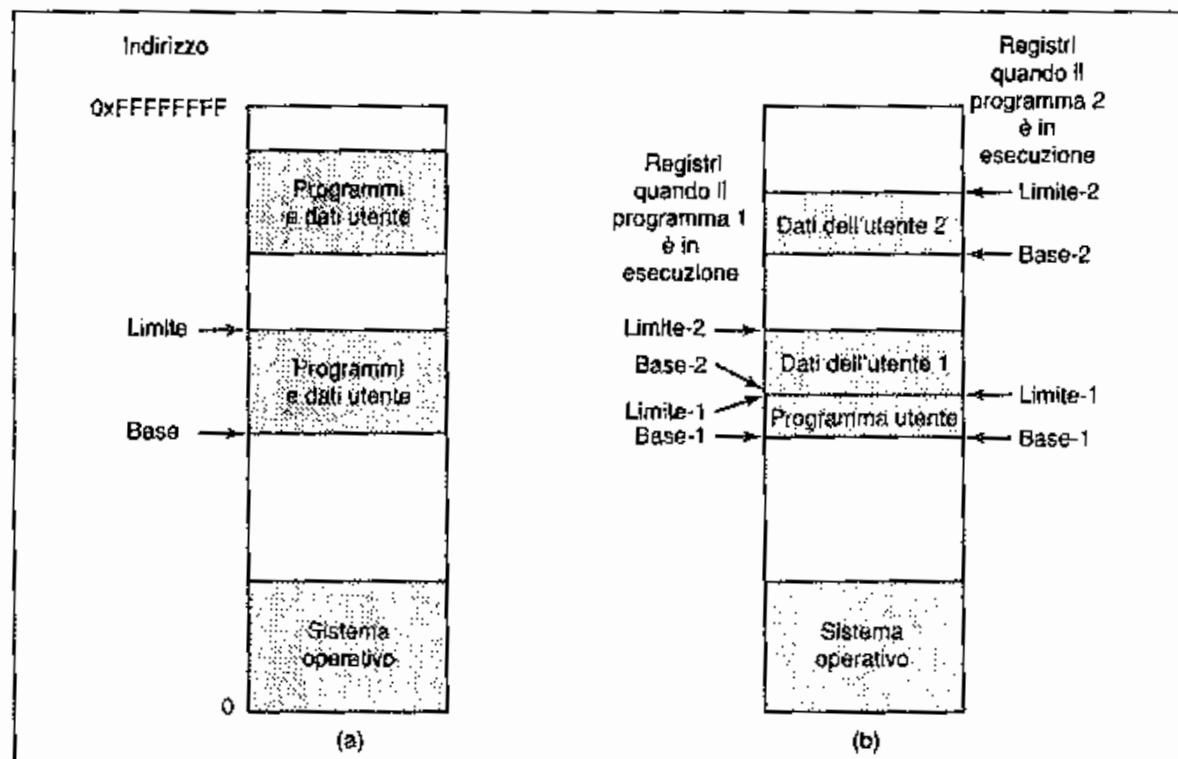


Figura 1.9 (a) Uso di una coppia base-limite. Il programma può accedere alla memoria tra la base e il limite. (b) L'uso di due coppie base-limite. Il codice del programma è compreso tra Base-1 e Limite-1 mentre i dati sono compresi fra Base-2 e Limite-2.

e il registro limite contiene il totale dello spazio occupato dal codice e dai dati del programma. Quando un'istruzione deve essere recuperata, l'hardware controlla che il program counter sia minore del registro limite e, se è così, lo somma al contenuto del registro base e spedisce il risultato alla memoria. Allo stesso modo, quando il programma vuole recuperare una parola di dati (per esempio dall'indirizzo 10000), l'hardware automaticamente somma il contenuto del registro base (per esempio 50000) a questo indirizzo, e spedisce il risultato (60000) alla memoria. Il registro base rende impossibile che il programma si riferisca a parti di memoria al di sotto di se stesso, e il registro limite rende impossibile che si riferisca a parti di memoria superiori al programma stesso. Questo schema risolve quindi entrambi i problemi di protezione e rilocazione, al costo di due nuovi registri e di un leggero aumento del tempo per l'esecuzione di un ciclo (per eseguire il controllo sul limite e la somma).

Il controllo e la corrispondenza si risolvono nella conversione dell'indirizzo generato dal programma, chiamato **indirizzo virtuale**, nell'indirizzo utilizzato dalla memoria, chiamato **indirizzo fisico**. Il dispositivo che effettua il controllo e la corrispondenza è detto **MMU (Memory Management Unit, unità per la gestione della memoria)** ed è collocato sul chip della CPU o vicino ad essa, ma, logicamente, sta tra la CPU e la memoria.

Una MMU più sofisticata è illustrata in Figura 1.9(b), in cui abbiamo una MMU con due coppie di registri base e limite, uno per il codice e uno per i dati. Il program counter e tutti gli altri riferimenti al codice del programma usano la coppia 1, e i riferimenti ai dati usano la coppia 2. Di conseguenza, più utenti possono condividere lo stesso programma tenendone una sola copia in memoria, cosa impossibile nel primo schema. Quando è in esecuzione il programma 1, i quattro registri sono impostati come indicato dalle frecce sulla sinistra della Figura 1.9(b); quando è in esecuzione il programma 2, sono impostati come indicato dalle frecce sulla destra della figura. Esistono MMU molto più sofisticate e studieremo alcune di esse, più avanti in questo libro. La cosa da notare qui è che la gestione della MMU deve essere compito del sistema operativo, dal momento che non è possibile fidarsi del fatto che gli utenti la svolgano correttamente.

Due aspetti del sistema di memoria hanno un effetto importante sulle prestazioni: primo, le cache nascondono la bassa velocità relativa della memoria. Quando un programma è in esecuzione da qualche tempo, la cache è piena delle linee di cache di quel programma, dando buone prestazioni, ma quando il sistema operativo passa da un programma a un altro, la cache rimane piena delle linee di cache del primo programma, e quelle necessarie al nuovo programma devono essere caricate, una alla volta, dalla memoria fisica. Questa operazione può dare un brutto colpo alle prestazioni, se viene effettuata troppo spesso.

Secondo, quando si passa da un programma all'altro, i registri della MMU devono essere modificati: in Figura 1.9(b), solo quattro registri devono essere reinpostati, il che non è un problema, ma nelle MMU vere devono essere ricaricati molti più registri, esplicitamente o dinamicamente, quando necessario; in entrambi i casi, occorre tempo. La morale della favola è che passare da un programma ad un altro, operazione chiamata **cambio di contesto** (context switch), è un affare costoso.

1.4.3 I dispositivi di ingresso/uscita

La memoria non è l'unica risorsa che deve essere gestita dal sistema operativo: anche i dispositivi di ingresso/uscita interagiscono pesantemente con esso. Come visto in Figura 1.5, i dispositivi di ingresso/uscita generalmente sono composti da due parti: un controllore e il dispositivo stesso. Il controllore è un chip (un circuito integrato) o un insieme di chip,

posto su una scheda estraibile, che controlla fisicamente il dispositivo, ed accetta comandi dal sistema operativo, per esempio, per leggere dati dal dispositivo, e li esegue.

In molti casi, il controllo reale del dispositivo è molto complicato e dettagliato, perciò è compito del controllore presentare un'interfaccia semplificata al sistema operativo. Per esempio, il controllore del disco può accettare un comando per leggere il settore 11206 dal disco 2; il controllore deve quindi convertire questo numero lineare di settore in un numero di cilindro, settore e testina. Questa conversione può essere complicata dal fatto che i cilindri esterni hanno più settori che quelli interni, e che alcuni settori danneggiati possono essere stati rimpiazzati da altri; quindi il controllore deve determinare su quale cilindro si trova il braccio del disco, e dargli una sequenza di impulsi perché si sposti all'interno o all'esterno del numero di cilindri necessari, poi deve aspettare fino a che il settore giusto è passato, ruotando, sotto la testina, e quindi può iniziare a leggere e memorizzare i bit come vengono dall'unità, eliminando il preambolo e calcolando il checksum (somma di controllo). Infine, deve assemblare i bit acquisiti in parole, e depositarli in memoria; per fare tutto questo lavoro, i controllori spesso contengono piccoli elaboratori embedded, che sono programmati per svolgere tale compito.

L'altro pezzo è il dispositivo vero e proprio. Le interfacce dei dispositivi sono veramente semplici, sia perché non sono in grado di fare molto, sia per renderle standard. Ciò è necessario per fare in modo che, per esempio, qualsiasi controllore di dischi IDE possa gestire qualunque disco IDE (Integrated Drive Electronics, elettronica per unità integrate), che è il tipo standard dei dischi sui Pentium e su altri elaboratori. Dal momento che le vere interfacce sono nascoste dietro al controllore, tutto ciò che il sistema operativo vede è l'interfaccia del controllore, che può essere abbastanza diversa dall'interfaccia del dispositivo.

Dato che ogni tipo di controllore è diverso, sono necessari software diversi per controllare ognuno di essi; il software che parla al controllore, dandogli i comandi e accettando le risposte, è detto **device driver** o **driver di dispositivo**, driver in breve. Ogni fabbricante di controllori deve fornire un driver per ogni sistema operativo che supporta, quindi, per esempio, con uno scanner possono essere forniti i driver per Windows 98, Windows 2000 e UNIX.

Per essere usato, un driver deve essere posto nel sistema operativo per essere eseguito in modalità kernel; in teoria, i driver possono essere eseguiti al di fuori del kernel, ma pochi sistemi attuali supportano questa possibilità, perché richiede la capacità, quasi mai supportata, di permettere ad un driver nello spazio utente di accedere ad un dispositivo in un modo controllato. Ci sono tre modi in cui un driver può essere messo nel kernel: il primo modo è effettuare nuovamente il link del kernel con il nuovo driver e riavviare il sistema, e molti sistemi UNIX funzionano in questo modo. Il secondo modo è effettuare una registrazione in un file di sistema dicendo che c'è bisogno del driver, e quindi riavviare il sistema; al momento del boot, il sistema operativo va a cercare i driver di cui necessita e li carica; un sistema che lavora così è Windows. Il terzo modo rende i sistemi operativi capaci di accettare nuovi driver durante l'esecuzione e di installarli al volo, senza bisogno di riavviare. Questo modo era raro, ma sta diventando molto comune: i dispositivi inseribili a macchina funzionante (hot pluggable device) come i dispositivi USB e IEEE 1394 (illustrati in seguito) hanno sempre bisogno di driver caricabili dinamicamente.

Ogni controllore ha un piccolo numero di registri che vengono usati per comunicare con esso. Per esempio, un controllore di disco minimale potrebbe avere registri che specificano l'indirizzo su disco, l'indirizzo di memoria, il numero di settori e la direzione (lettura o scrittura); per attivare il controllore, il driver riceve un comando dal sistema operativo, quindi lo traduce nei valori appropriati da scrivere nei registri del dispositivo.

Su qualche elaboratore, i registri di dispositivo sono messi in corrispondenza con indirizzi all'interno dello spazio di indirizzamento del sistema operativo, in modo che possa-

no essere letti e scritti come parole di memoria normali; su tali elaboratori, non sono necessarie istruzioni speciali per l'ingresso/uscita e i programmi utente possono essere tenuti lontani dall'hardware non mettendo questi indirizzi di memoria entro il loro raggio di azione (per esempio usando i registri base e limite). Su altri elaboratori, i registri di dispositivo sono messi in uno spazio speciale per le porte di ingresso/uscita, con ogni registro che corrisponde ad un indirizzo di porta. Su queste macchine, in modalità kernel sono disponibili le istruzioni speciali IN e OUT, per permettere ai driver di leggere e scrivere i registri. Il primo modo elimina il bisogno di istruzioni speciali per l'ingresso/uscita, ma usa una parte dello spazio di indirizzamento, mentre il secondo non usa spazio di indirizzamento ma richiede istruzioni speciali; entrambi i sistemi sono largamente utilizzati.

L'ingresso e l'uscita possono essere effettuati in tre modi diversi: nel metodo più semplice, un programma utente emette una chiamata di sistema, che il kernel traduce in una chiamata di procedura al driver appropriato; dopodiché il driver dà il via all'operazione di ingresso/uscita e entra in un ciclo serrato, durante il quale continua a interrogare il dispositivo per verificare se questo ha terminato l'operazione (generalmente dei bit indicano se il dispositivo è ancora occupato). Quando l'operazione di ingresso/uscita è completata, il driver restituisce i dati richiesti, se ce ne sono, e termina; il sistema operativo quindi restituisce il controllo al chiamante. Questo metodo è detto **busy waiting** (attesa attiva) e ha lo svantaggio di tenere occupata la CPU nell'interrogazione del dispositivo (polling) finché l'operazione di ingresso/uscita è finita.

Il secondo metodo prevede che il driver faccia partire il dispositivo, e che questo mandi una interruzione al driver quando ha finito, dopodiché il driver termina la sua esecuzione. Il sistema operativo, nel frattempo, blocca il programma chiamante, se è necessario, e controlla se ci sono altri lavori da fare. Quando il controllore rileva la fine di un trasferimento, genera una **interruzione** (interrupt) per segnalare il completamento dell'operazione.

Le interruzioni sono molto importanti nei sistemi operativi, di conseguenza esaminiamo l'idea più da vicino. Nella Figura 1.10(a) vediamo un processo di ingresso/uscita a tre passi; nel primo passo, il driver dice al controllore cosa fare scrivendo nei suoi registri di dispositivo, dopodiché il controllore fa partire il dispositivo. Quando il controllore ha

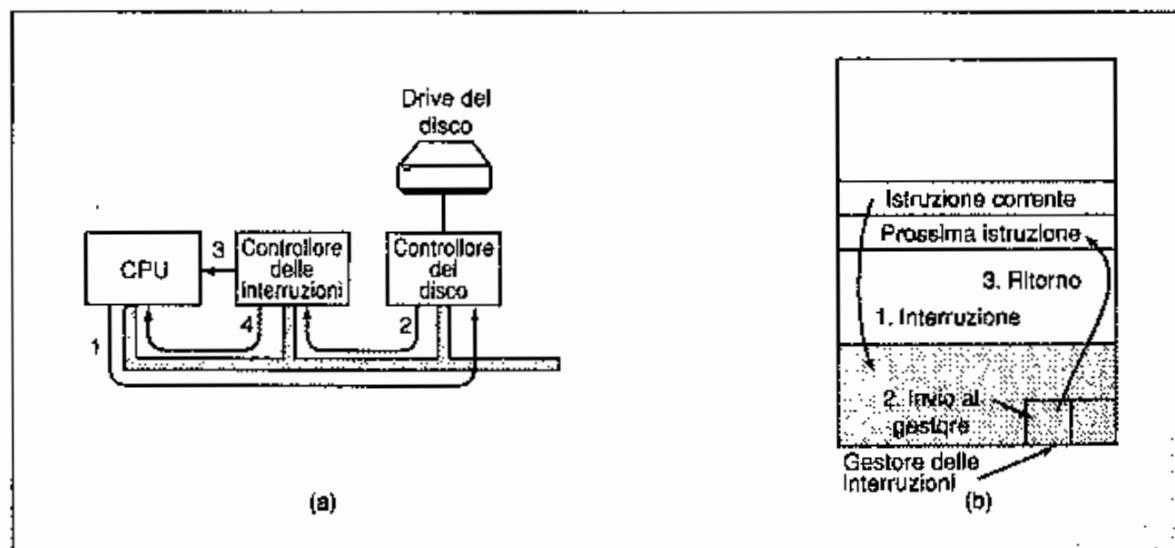


Figura 1.10 (a) I passi di esecuzione nell'avvio di un dispositivo di ingresso/uscita e nella ricezione di un'interruzione. (b) La gestione di un'interruzione comporta la ricezione dell'interruzione, l'esecuzione del gestore delle interruzioni e il ritorno al programma utente.

finito di leggere o di scrivere il numero di byte che gli è stato detto di trasferire, lo segnala al controllore dell'interruzione, usando ben precise linee del bus (passo 2). Se il controllore dell'interruzione si aspetta l'interruzione (il che non può essere se è occupato con una interruzione a priorità più alta), al passo 3 informa la CPU, impostando un segnale. Al passo 4, il controllore dell'interruzione mette il numero del dispositivo sul bus, in modo che la CPU possa leggerlo e sapere quindi quale dispositivo ha appena finito (molti dispositivi possono essere in esecuzione contemporaneamente).

Una volta che la CPU ha deciso di accettare una interruzione, generalmente il program counter e il PSW vengono messi sulla pila corrente e la CPU passa alla modalità kernel. Il numero di dispositivo può essere usato come indice per cercare, all'interno di una parte della memoria, l'indirizzo del gestore di interruzioni per quel dispositivo; questa parte di memoria è chiamata **vettore delle interruzioni** (interrupt vector). Una volta che è partito il gestore delle interruzioni, che fa parte del driver del dispositivo che ha mandato l'interruzione, esso rimuove dallo stack il program counter e il PSW e li salva, dopodiché interroga il dispositivo per conoscere il suo stato. Quando il gestore ha terminato, il controllo ritorna alla prima istruzione non ancora eseguita nel programma utente precedentemente in esecuzione; questi passi sono mostrati in Figura 1.10(b).

Il terzo metodo per fare ingresso/uscita fa uso di un chip speciale, detto DMA (Direct Memory Access, accesso diretto in memoria), che può controllare il flusso dei bit tra la memoria e i controllori senza il continuo intervento della CPU. La CPU avvia il chip DMA, specificando quanti bit deve trasferire, il dispositivo e gli indirizzi di memoria coinvolti, la direzione e così via; quando il chip DMA ha terminato, genera una interruzione, che viene gestita come descritto sopra. Il DMA e l'hardware di ingresso/uscita in generale verranno discussi più in dettaglio nel Cap. 5.

Le interruzioni possono verificarsi in momenti altamente inopportuni, ad esempio, quando è in esecuzione un altro gestore di interruzioni, per questo motivo, la CPU ha la possibilità di disabilitare le interruzioni e riabilitarle in seguito. Mentre le interruzioni sono disabilitate, ogni dispositivo che termina continua a inviare segnali di interruzione, ma la CPU non viene interrotta finché le interruzioni non vengono di nuovo riabilitate. Se più dispositivi finiscono mentre le interruzioni sono disabilitate, il controllore delle interruzioni decide quale deve passare per primo, generalmente basandosi su priorità statiche assegnate a ciascun dispositivo: il dispositivo a più alta priorità vince.

1.4.4 I bus

L'organizzazione di Figura 1.5 è stata usata sui minicalcolatori per anni e anche sul PC IBM originale. Comunque, mentre i processori e le memorie si evolvevano in fretta, l'abilità di un singolo bus (e certamente del bus dei PC IBM) di manovrare tutto il traffico è stata forzata fino al punto di rottura. Qualcosa andava fatto, così vennero aggiunti altri bus, sia per dispositivi di ingresso/uscita più veloci, sia per il traffico tra CPU e memoria. Come conseguenza di questa evoluzione, un sistema Pentium, attualmente, sembra più simile a quanto mostrato in Figura 1.11.

Questo sistema ha otto bus (cache, memoria, locale, PCI, SCSI, USB, IDE e ISA), ciascuno con una diversa velocità di trasferimento e diverse funzioni: il sistema operativo deve essere consapevole di ciascuno di essi per configurarli e gestirli. I due bus principali sono quello originale del PC IBM, il bus ISA (Industry Standard Architecture) e il suo successore, il bus PCI (Peripheral Component Interconnect). Il bus ISA, originariamente il bus del PC/AT di IBM, lavora a 8,33 MHz e può trasferire 2 byte per volta, per una velocità massima di 16,67 MB/sec; è presente per mantenere la compatibilità con schede di ingresso/uscita vecchie e lente. Il bus PCI venne inventato da Intel come successore

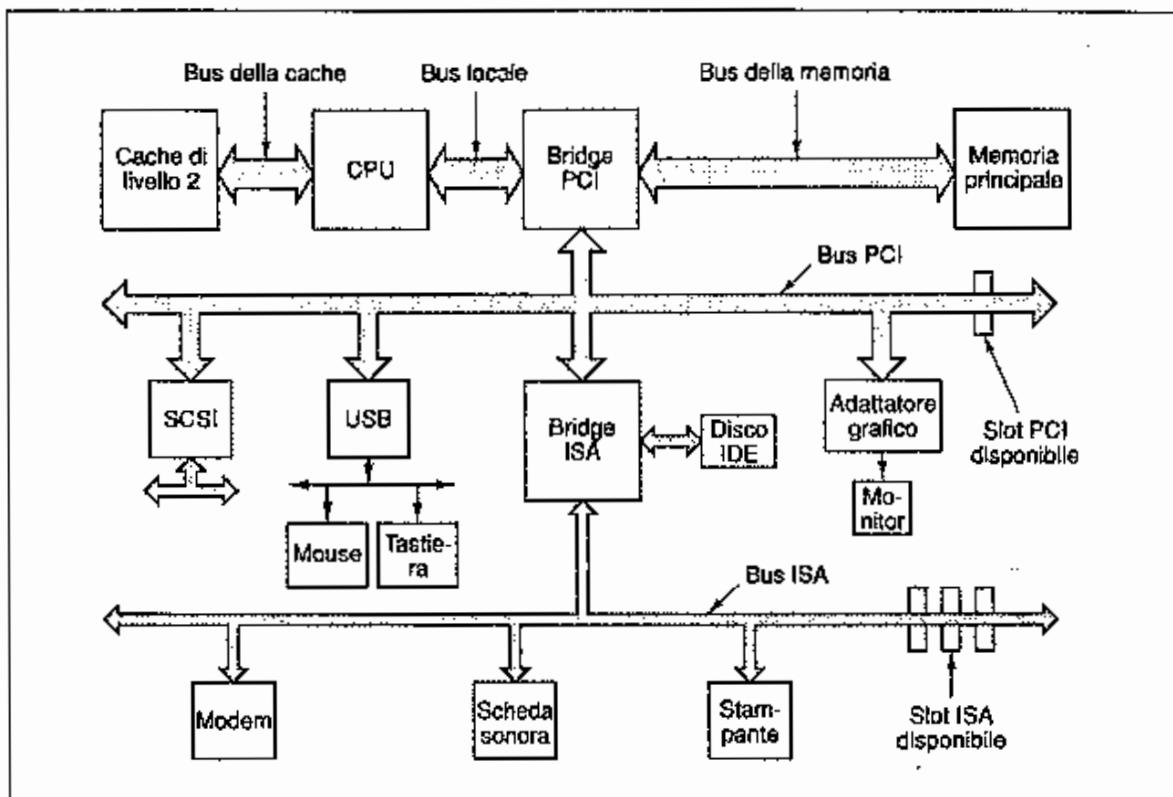


Figura 1.11 La struttura di un grosso sistema Pentium.

del bus ISA, può lavorare a 66MHz e trasferire 8 byte per volta, per una velocità di trasferimento dei dati (data rate) di 528 MB/s. La maggior parte dei dispositivi di ingresso/uscita odierni usa il bus PCI, di conseguenza persino qualche calcolatore non Intel lo usa, a causa del gran numero di schede di I/O disponibili.

In questa configurazione, la CPU comunica tramite il bus locale con il chip PCI bridge, e questo comunica con la memoria tramite un bus dedicato, spesso un bus a 100 MHz. I sistemi Pentium hanno una cache di livello 1 su chip, e una cache più grande di livello 2 all'esterno del chip, collegata alla CPU dal bus di cache.

In aggiunta, questo sistema contiene tre bus specializzati: IDE, USB e SCSI. Il bus IDE serve per il collegamento al sistema di periferiche come dischi e CD-ROM. Il bus IDE è il discendente del controllore del disco del PC/AT, ed è ora uno standard su quasi tutti i sistemi basati su processore Pentium per i dischi fissi e, spesso, anche per i CD-ROM.

L'**USB** (Universal Serial Bus) è stato inventato per permettere di collegare al calcolatore tutti i dispositivi di ingresso/uscita lenti, come la tastiera e il mouse. Utilizza un piccolo connettore a quattro fili, due dei quali forniscono energia elettrica ai dispositivi USB; USB è un bus centralizzato nel quale un dispositivo radice interroga i dispositivi di ingresso/uscita una volta ogni millisecondo per verificare se hanno un traffico di qualche tipo; esso può gestire un carico complessivo di 1,5 MB/s. Tutti i dispositivi USB condividono un solo driver di dispositivo USB, evitando di dover installare un nuovo driver per ogni dispositivo USB. Di conseguenza, i dispositivi USB possono essere collegati senza la necessità di riavviare il calcolatore.

Il bus **SCSI** (Small Computer System Interface) è un bus a alte prestazioni destinato a dischi veloci, scanner e altri dispositivi che necessitano di una considerevole larghezza di banda, che può arrivare fino a 160MB/s. È presente sui sistemi Macintosh da quando

furono inventati, ed è popolare anche sui sistemi UNIX e su qualche sistema basato su processore Intel.

Un altro bus (non mostrato in Figura 1.11) è l'**IEEE 1394**, qualche volta detto FireWire, sebbene, per essere precisi, FireWire sia il nome che Apple ha dato alla propria implementazione del 1394. Come l'USB, IEEE 1394 è seriale, ma è progettato per trasferimenti a pacchetti alla velocità massima di 50 MB/s, rendendolo utile per il collegamento al calcolatore di telecamere digitali e altri dispositivi multimediali simili. Diversamente dall'USB, IEEE 1394 non ha un controllore centrale; SCSI e IEEE 1394 hanno un competitore nella versione di USB più veloce che si sta sviluppando.

Per lavorare in un ambiente come quello mostrato in Figura 1.11, il sistema operativo deve conoscere cosa c'è al di fuori e configurarlo; questo requisito ha portato Intel e Microsoft a progettare un sistema per PC chiamato **plug and play** (letteralmente, "attacca e gioca"), basato su un concetto simile già implementato su Macintosh di Apple. Prima del plug and play, ogni scheda di ingresso/uscita doveva avere un livello di richiesta di interruzione e indirizzi fissi per i suoi registri di ingresso/uscita. Per esempio, la tastiera aveva l'interruzione 1 e usava gli indirizzi di ingresso/uscita da 0x60 a 0x64, il controllore del floppy aveva l'interruzione 6 e usava gli indirizzi di ingresso/uscita da 0x3F0 a 0x3F7 e la stampante aveva livello di interruzione 7 e usava gli indirizzi di ingresso/uscita da 0x378 a 0x37A e così via.

Fin qui tutto bene. Il brutto arrivava quando l'utente comprava una scheda sonora e un modem e capitava che entrambi usassero, diciamo, l'interruzione 4: andavano in conflitto e non potevano lavorare insieme. La soluzione fu di includere degli interruttori DIP o dei jumper su ogni scheda di ingresso/uscita, e istruire l'utente a posizionarli in modo da selezionare un livello di interruzione e indirizzi di dispositivo di ingresso/uscita che non andassero in conflitto con gli altri nel sistema dell'utente. I ragazzini che dedicavano la loro vita alle complicazioni dell'hardware del PC riuscirono qualche volta a farlo senza commettere errori; sfortunatamente, nessun altro ci riusciva, il che portò al caos.

Il plug and play permette al sistema di raccogliere automaticamente le informazioni relative ai dispositivi di ingresso/uscita, assegnare in modo centralizzato i livelli di interruzione e gli indirizzi di ingresso/uscita, e comunicare ad ogni scheda il proprio numero. Brevemente, ecco come questo avviene nel Pentium: ogni Pentium contiene una scheda genitore (in passato detta scheda madre, prima che la political correctness raggiungesse l'industria dei calcolatori), sulla quale risiede un programma detto BIOS (**Basic Input Output System**), che contiene software di basso livello per l'ingresso/uscita, comprese le procedure per leggere la tastiera, scrivere sul video e fare ingresso/uscita su disco. Oggigiorno, è contenuto in una flash RAM, che è non volatile e può essere aggiornata dal sistema operativo quando vengono rilevati dei bachi nel BIOS.

Quando il calcolatore viene acceso, viene lanciato il BIOS, che prima verifica quanta RAM è installata e se la tastiera e altri dispositivi di base sono presenti e rispondono correttamente. Comincia ad analizzare i bus ISA e PCI, per determinare tutti i dispositivi collegati ad essi; alcuni di questi sono tipicamente **legacy** (letteralmente, eredità, cioè progettati prima che venisse inventato il plug and play) e hanno livelli di interruzione e indirizzi di ingresso/uscita fissi (impostabili da interruttori o jumper posti sulla scheda di ingresso/uscita, ma non modificabili da sistema operativo). Questi dispositivi vengono registrati, così come i dispositivi plug and play; se i dispositivi presenti sono diversi dall'ultima volta che il sistema è stato inizializzato, quelli nuovi vengono configurati.

Il BIOS quindi determina il dispositivo di memoria da cui effettuare l'inizializzazione (boot) provando i dispositivi elencati in una lista memorizzata nella memoria CMOS, che l'utente può modificare eseguendo il configuratore del BIOS immediatamente dopo aver acceso il calcolatore. Generalmente, viene effettuato un tentativo di inizializzazione dal dischetto; se questo fallisce, si prova dal CD-ROM; se non sono presenti né un floppy né

un CD-ROM, il sistema viene inizializzato dal disco fisso. Viene caricato in memoria ed eseguito il primo settore del dispositivo di inizializzazione, che contiene un programma che, generalmente, esamina la tabella delle partizioni posta alla fine del settore di inizializzazione e determina quale partizione è attiva. Da questa viene caricato in memoria un programma di inizializzazione secondario, che carica in memoria il sistema operativo, posto nella partizione attiva, e lo fa partire.

Il sistema operativo interroga il BIOS per avere le informazioni di configurazione, e per ogni dispositivo, controlla se ha il driver. Se non lo ha, chiede all'utente di inserire un floppy o un CD-ROM contenente il driver (fornito dal produttore del dispositivo); ed una volta che ha tutti i driver di dispositivo, il sistema operativo li carica nel kernel, dopodiché inizializza le sue tabelle, crea i processi di background necessari ed esegue un programma di login o una GUI su ogni terminale. O almeno questo è il modo in cui si suppone che lavori; nella realtà, il plug and play è spesso così inaffidabile che molte persone lo chiamano plug and pray (letteralmente, "attacca e prega").

1.5 Concetti di base sui sistemi operativi

Tutti i sistemi operativi hanno determinati concetti di base, come processi, memoria e file, che sono determinanti per capirli. Nelle sezioni seguenti, daremo un'occhiata ad alcuni di questi concetti basilari, molto brevemente, come introduzione; torneremo sui dettagli di ciascuno di essi più avanti nel libro. Per descrivere questi concetti useremo di volta in volta degli esempi, normalmente presi da UNIX; comunque, esempi simili esistono generalmente anche per altri sistemi operativi.

1.5.1 I processi

Il processo è un concetto chiave in tutti i sistemi operativi: un processo è, essenzialmente, un programma in esecuzione. Ad ogni processo vengono associati il suo spazio di indirizzamento (address space) e una lista di locazioni di memoria, da un qualche minimo (generalmente 0) a un qualche massimo, che il processo può leggere e scrivere. Lo spazio di indirizzamento contiene il programma eseguibile, i dati del programma e il suo stack (pila). Inoltre, ad ogni processo viene associato un insieme di registri, tra cui il suo program counter (contatore di programma), lo stack pointer (puntatore alla pila) e altri registri hardware, e tutte le altre informazioni necessarie all'esecuzione del programma.

Rituneremo sul concetto di processo, con maggiori dettagli, nel Capitolo 2, ma per il momento, il modo più semplice per avere una buona idea intuitiva di processo è di pensare ai sistemi timesharing. Periodicamente, il sistema operativo decide di sospendere l'esecuzione di un processo e di iniziare ad eseguirne un altro perché, ad esempio, il primo ha esaurito il tempo di CPU che aveva a disposizione nel secondo precedente.

Quando un processo viene temporaneamente sospeso, come in questo caso, più tardi deve essere fatto ripartire esattamente nello stato in cui era prima della sospensione. Questo significa che tutte le informazioni relative al processo devono essere esplicitamente salvate da qualche parte durante la sospensione. Ad esempio, il processo potrebbe avere diversi file aperti in lettura contemporaneamente; a ciascuno di questi file è associato un puntatore che dà la posizione corrente all'interno del file (cioè, il numero del byte o del record successivo da leggere). Quando un processo viene temporaneamente sospeso, tutti questi puntatori vanno salvati, in modo che una successiva operazione di read, eseguita dopo che il processo è ripartito, possa leggere i dati corretti. In molti sistemi operativi,

tutte le informazioni su ciascun processo, tranne il contenuto del suo spazio di indirizzamento, sono memorizzate in una tabella di sistema operativo chiamata **tabella dei processi** (*process table*), che è un array (o una lista) di strutture, una per ogni processo attualmente esistente.

Quindi un processo (eventualmente sospeso) è costituito dal suo spazio di indirizzamento, solitamente chiamato **immagine in memoria** (*core image*, in onore delle memorie a nuclei magnetici che si usavano una volta) e dal corrispondente elemento della tabella dei processi, che contiene, tra l'altro, i suoi registri.

Le principali chiamate di sistema per la gestione dei processi sono quelle che si occupano della creazione e della terminazione dei processi. Consideriamo un esempio tipico: un processo chiamato **interprete di comandi**, o **shell**, legge comandi da un terminale; l'utente ha appena battuto un comando richiedendo la compilazione di un programma; la shell deve ora creare un nuovo processo che eseguirà il compilatore; quando quel processo avrà concluso la compilazione, eseguirà una chiamata di sistema per terminarsi.

Se un processo può creare uno o più altri processi, detti **processi figli** (*child processes*), e questi processi possono a loro volta creare processi figli, arriviamo velocemente all'albero dei processi di Figura 1.12. Processi collegati che cooperano delegando del lavoro ad altri spesso hanno bisogno di comunicare fra loro e sincronizzare le loro attività. Questa comunicazione è detta **comunicazione fra processi** (*interprocess communication*) a cui faremo riferimento dettagliatamente nel Capitolo 2.

Sono disponibili altre chiamate di sistema relative ai processi, che consentono di richiedere più memoria (o restituire memoria non utilizzata), attendere la terminazione di un processo figlio e sovrapporre un altro programma al proprio.

Talvolta, sorge la necessità di inviare informazioni ad un processo in esecuzione che in quel momento non le sta aspettando; ad esempio, un processo che sta comunicando con un altro processo su un altro calcolatore, lo fa spedendo messaggi al processo remoto su una rete. Per prevenire la possibilità che un messaggio o la risposta ad esso vengano persi, il mittente può richiedere che il suo sistema operativo lo avvisi dopo un certo numero di secondi, in modo da poter ritrasmettere il messaggio nel caso in cui non abbia ancora ricevuto nessun messaggio di acknowledgement (conferma dell'avvenuta ricezione). Dopo aver fatto partire questo timer, il programma può continuare a svolgere un altro lavoro.

Quando il tempo specificato è trascorso, il sistema operativo spedisce un **segnale di allarme** al processo, che causa una sospensione momentanea delle attività del processo, qualsiasi cosa questo stesse facendo, il salvataggio dei suoi registri sullo stack e l'inizio dell'esecuzione di una procedura particolare di trattamento del segnale, che si occuperà, ad esempio, di ritrasmettere il messaggio che presumibilmente è andato per-

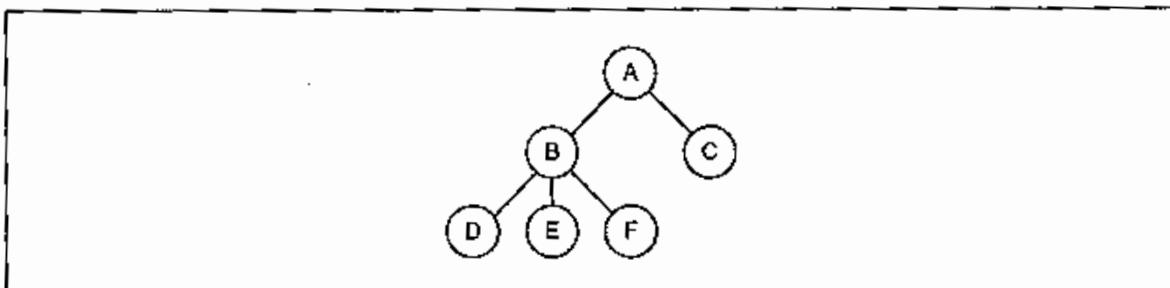


Figura 1.12 Un albero di processi. Il processo A ha creato due processi figli, B e C. Il processo B ha creato tre processi figli, D, E e F.

duto. Quando la procedura di trattamento del segnale è terminata, il processo che era in esecuzione viene fatto ripartire nello stato in cui si trovava immediatamente prima della ricezione del segnale. I segnali sono il corrispondente software delle interruzioni hardware e possono essere generati da molte cause, oltre che dallo scadere di un timer. Anche molte delle interruzioni generate da situazioni irregolari (trap) rilevate dall'hardware, come per esempio l'esecuzione di una istruzione illegale o il riferimento ad un indirizzo non valido, vengono convertite in segnali indirizzati al processo responsabile della situazione.

Ad ogni persona autorizzata all'uso di un sistema (utente) l'amministratore del sistema stesso assegna un **UID** (User Identification, identificatore di utente). Ogni processo che viene eseguito ha l'UID della persona che l'ha lanciato, un processo figlio ha lo stesso UID del processo che l'ha generato. Gli utenti possono essere membri di gruppi, ognuno dei quali ha un **GID** (Group Identification, identificatore di gruppo).

Un UID, chiamato **superuser** (in UNIX), ha poteri speciali e può violare la maggior parte delle regole di protezione. In grossi sistemi, solo l'amministratore di sistema conosce la password necessaria a diventare superuser, ma molti degli utenti comuni (in particolar modo gli studenti) dedicano sforzi considerevoli al tentativo di trovare errori nel sistema che permettano loro di diventare superuser senza password.

Studieremo i processi, la comunicazione fra processi e i problemi correlati nel Capitolo 2.

1.5.2 Il deadlock

Quando due o più processi interagiscono, a volte possono mettersi in una situazione di stallo dalla quale non possono uscire, che viene chiamata **deadlock** (stallo).

Il deadlock può essere meglio illustrato con un esempio di vita reale familiare a ciascuno, il deadlock nel traffico. Consideriamo la situazione di Figura 1.13(a), nella quale quattro autobus si stanno avvicinando ad un incrocio; dietro a ciascun autobus ce ne sono altri (non disegnati). Con un pizzico di sfortuna, i primi quattro arrivano contemporaneamente all'incrocio, portando alla situazione mostrata in Figura 1.13(b), nella quale sono tutti irrimediabilmente bloccati, perché nessuno può più proseguire: ciascun autobus blocca uno degli altri, e nessuno può retrocedere a causa degli autobus che li seguono. Non c'è una facile via d'uscita.

I processi di un calcolatore possono incorrere in una situazione analoga, nella quale

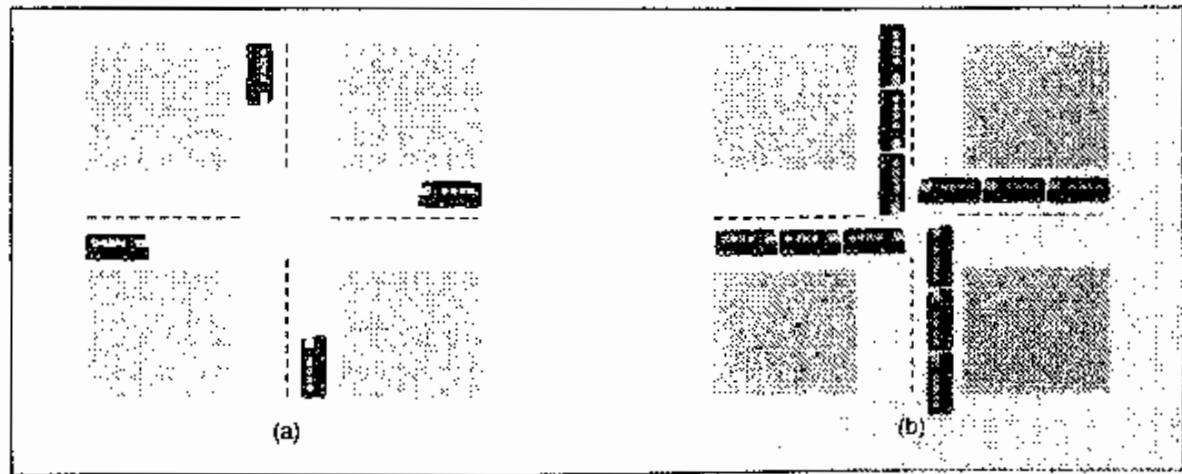


Figura 1.13 [a] Un deadlock potenziale. [b] Un deadlock reale.

non possono proseguire. Ad esempio, immaginiamo un calcolatore con un drive a nastro e un masterizzatore CD, ed immaginiamo che due processi abbiano bisogno di creare un CD-ROM dai dati presenti sul nastro. Il processo 1 richiede e ottiene il nastro, poi il processo 2 richiede e ottiene l'uso del masterizzatore; il processo 1 richiede il masterizzatore e viene sospeso finché il processo 2 non lo restituisce; infine, il processo 2 richiede il nastro e viene sospeso perché la risorsa è già occupata dal processo 1: abbiamo un deadlock dal quale non c'è via d'uscita. Studieremo in dettaglio i deadlock e cosa può essere fatto in merito nel Capitolo 3.

1.5.3 La gestione della memoria

Ogni calcolatore ha della memoria principale nella quale si trovano i programmi in esecuzione. In un sistema operativo molto semplice, in memoria c'è solo un programma per volta e, per eseguire un secondo programma, bisogna rimuovere il primo, e poi il secondo può essere caricato in memoria.

Sistemi operativi più sofisticati permettono di mantenere più programmi in memoria contemporaneamente. Per evitare che interferiscano l'un l'altro (e con il sistema operativo), è necessario un qualche meccanismo di protezione; tale meccanismo deve essere nell'hardware, ma deve essere controllato dal sistema operativo.

Il punto di vista di cui sopra si occupa di manipolare e proteggere la memoria principale del calcolatore. Un problema diverso, ma ugualmente importante, legato alla memoria è la gestione dello spazio di indirizzamento dei processi: di solito, ogni processo ha un insieme di indirizzi che può usare, che generalmente va da 0 a un qualche massimo. Nel caso più semplice, la dimensione massima di spazio di indirizzamento di un processo è minore della dimensione della memoria principale, così un processo può riempire il suo spazio di indirizzamento e ci sarà ancora abbastanza spazio in memoria principale per contenerlo tutto.

Comunque, su molti calcolatori gli indirizzi sono di 32 o 64 bit, che consentono uno spazio di indirizzi rispettivamente di 2^{32} o 2^{64} byte. Cosa accade se un processo ha uno spazio di indirizzamento più grande della memoria principale e lo vuole usare tutto? Nei primi calcolatori, un processo di questo tipo era semplicemente sfortunato, ma, oggi giorno, esiste una tecnica chiamata memoria virtuale, che fa sì che il sistema operativo gestisca lo spazio di indirizzamento mantenendone una parte in memoria principale e una parte su disco, trasportando dei frammenti avanti e indietro tra questi secondo quanto necessario. Questa funzione importante del sistema operativo, e altre funzioni legate alla memoria, saranno trattate nel Capitolo 4.

1.5.4 L'ingresso/uscita

Tutti i calcolatori hanno dispositivi fisici per acquisire dati in ingresso e per produrre dati in uscita; dopo tutto, quanto sarebbe utile un calcolatore, se l'utente non potesse dirgli cosa fare e non potesse ottenere i risultati dopo che il lavoro richiesto è stato svolto? Esistono molti tipi di dispositivi di ingresso e uscita, tra cui tastiere, monitor, stampanti e così via, ed è compito del sistema operativo gestirli.

Di conseguenza, ogni sistema operativo ha un sottosistema di ingresso/uscita per gestire i dispositivi di ingresso/uscita. Una parte del software di ingresso/uscita è indipendente dal dispositivo, cioè, può essere applicata a molti o a tutti i dispositivi di ingresso/uscita in modo equivalente, ma un'altra parte, come i driver di dispositivo, è specifica di particolari dispositivi di ingresso/uscita. Nel Capitolo 5 daremo un'occhiata al software di ingresso/uscita.

1.5.5 I file

Un altro concetto supportato di fatto da tutti i sistemi operativi è il file system. Come già messo in evidenza precedentemente, una delle funzioni principali del sistema operativo è quella di mascherare le particolarità dei dischi e degli altri dispositivi di ingresso/uscita, e presentare al programmatore un modello pulito ed astratto di file indipendente dai dispositivi. Ovviamente, le chiamate di sistema sono necessarie per creare, rimuovere, leggere o scrivere file; prima di poter leggere un file occorre localizzarlo sul disco e aprirlo, e dopo averlo letto occorre chiuderlo, così che sono necessarie chiamate di sistema per provvedere a questi scopi.

Per fornire un posto dove tenere i file, la maggior parte dei sistemi operativi supporta il concetto di directory come mezzo per raggruppare i file. Uno studente, ad esempio, potrebbe avere una directory per ognuno dei corsi che sta seguendo (in cui conserva i programmi necessari al corso), un'altra per la posta ed un'altra ancora per la sua pagina Web. Per creare e rimuovere directory, sono necessarie opportune chiamate di sistema, così come servono chiamate per inserire e per rimuovere un file da una directory. Gli elementi di una directory possono essere sia file che directory: questo modello dà vita ad una gerarchia: il file system, come mostrato in Figura 1.14.

Le gerarchie di processi e di file sono entrambe organizzate come alberi, ma le somiglianze si fermano qui. Le gerarchie di processi non sono in generale molto profonde (di solito non vanno oltre i tre livelli), mentre sono comuni gerarchie di file con quattro, cinque ed anche un numero maggiore di livelli di profondità. Le gerarchie di processi hanno di solito una vita breve, al massimo di qualche minuto, mentre le gerarchie di directory esistono per anni. Anche i concetti di appartenenza e di protezione sono diversi: soli-

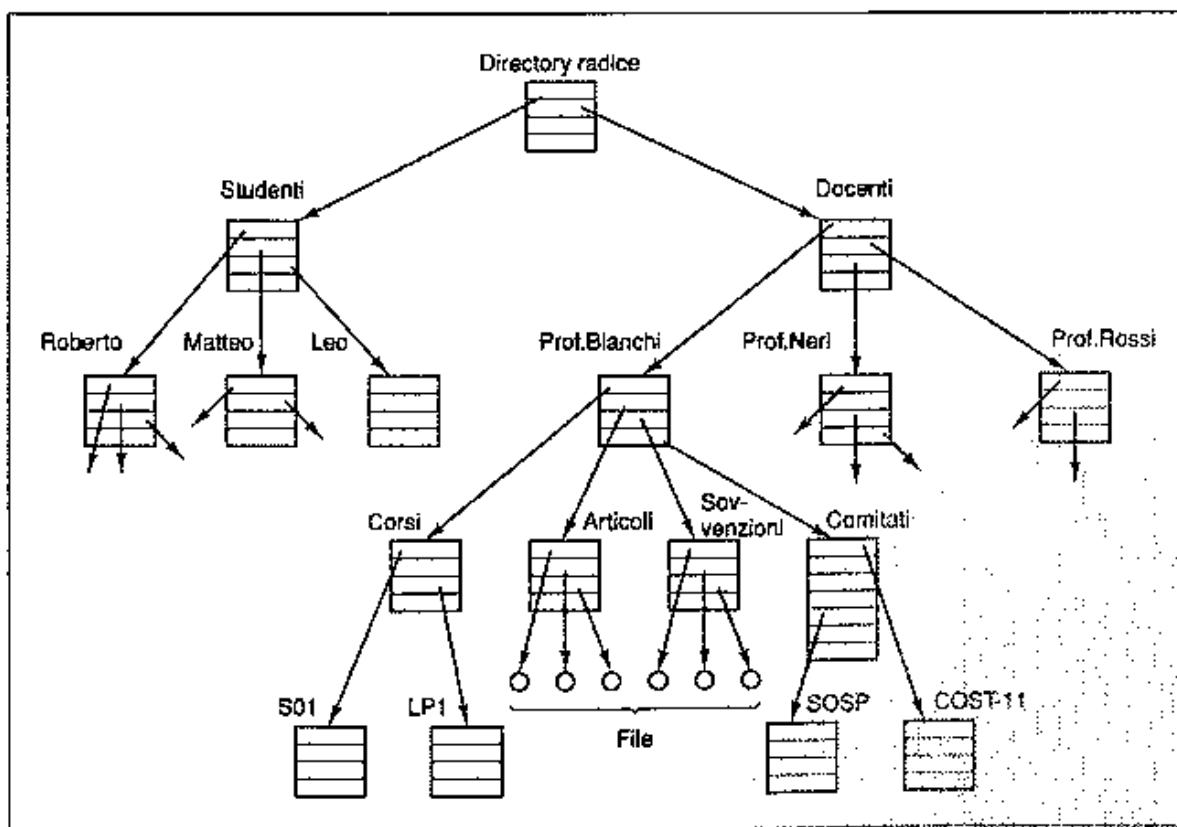


Figura 1.14 Un file system per un dipartimento dell'università.

tamente, solo il processo padre può controllare o persino accedere ad un processo figlio, mentre quasi sempre esistono meccanismi che permettono la lettura di file e directory a gruppi più ampi di quelli composti dal solo proprietario.

Ciascun file all'interno della gerarchia può essere specificato indicandone il **path name** (cammino) a partire dalla cima della gerarchia, detta **directory radice** (**root directory**). Il cammino assoluto è formato dalla serie di directory che devono essere attraversate a partire dalla directory radice per arrivare al file, con slash (" / ", la barra normalmente utilizzata come simbolo della divisione) che dividono tali componenti. In Figura 1.14, il path name per il file SO1 è */Docenti/Prof.Bianchi/Corsi/SO1*. La barra iniziale indica che il path name è assoluto, cioè che parte dalla directory radice. Fra parentesi, in MS-DOS e in Windows, è utilizzato il carattere backslash (\) come separatore invece del carattere slash (/), perciò il path name per il file dato prima dovrebbe essere scritto come *\Docenti\Prof.Bianchi\Corsi\SO1*. Nel corso del libro, di solito, per i path name useremo la convenzione di UNIX.

Ad ogni istante, ciascun processo ha una **directory di lavoro** (**working directory**) corrente, nella quale vengono ricercati i nomi che non iniziano con uno slash. In Figura 1.14, se la directory corrente fosse */Docenti/Prof.Bianchi*, l'uso del nome *Corsi/SO1* porterebbe allo stesso file individuato dal path name assoluto riportato sopra. I processi possono cambiare la loro directory di lavoro eseguendo una chiamata di sistema, e specificando la nuova directory di lavoro.

Prima che un file venga letto o scritto, esso deve essere aperto, ed è in quel momento che vengono controllati i diritti d'accesso. Se l'accesso è permesso, il sistema restituisce un numero intero, normalmente molto piccolo, chiamato **descrittore di file** (**file descriptor**), che verrà poi usato nelle operazioni successive; se l'accesso è proibito, viene restituito un codice di errore.

Un altro concetto importante di UNIX è quello di **file system** montato: quasi tutti i personal computer hanno uno o più drive dei dischetti nel quale i dischetti possono essere inseriti o rimossi. Per fornire un modo elegante di avere a che fare con supporti rinnovabili, (tra cui i CD-ROM) UNIX permette al file system di un dischetto di venire attaccato all'albero principale. Si consideri la situazione di Figura 1.15(a): prima della chiamata di **mount**, il **file system radice** (**root file system**) sul disco fisso, e un secondo file system, su dischetto, sono separati e scollegati.

Comunque, il file system sul dischetto non può essere usato, perché non c'è modo di specificare cammini su di esso: UNIX non permette di specificare cammini preceduti da un nome di drive o da un numero; questo sarebbe precisamente il tipo di dipendenza da dispositivo che il sistema operativo dovrebbe eliminare. Invece, la chiamata di sistema

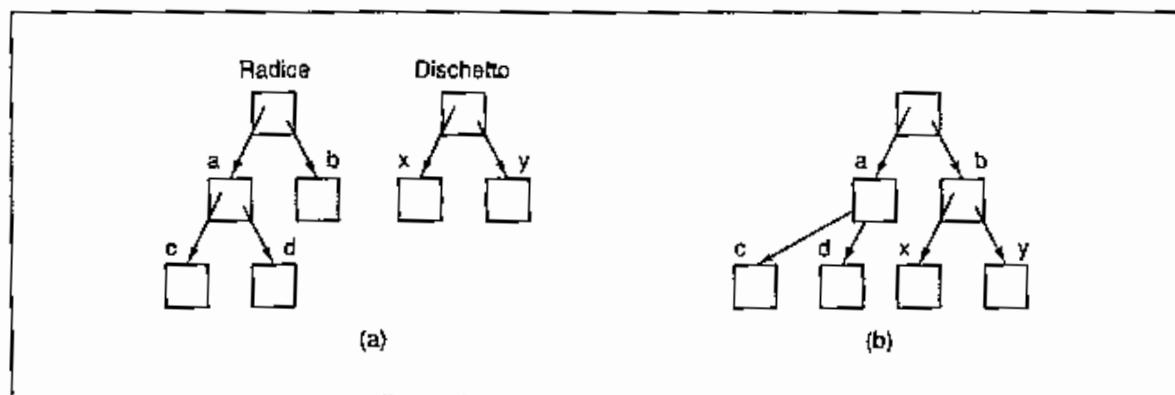


Figura 1.15 (a) Prima del mount, i file sul drive 0 (dischetto) non sono accessibili. (b) Dopo il mount, sono parte della gerarchia dei file.

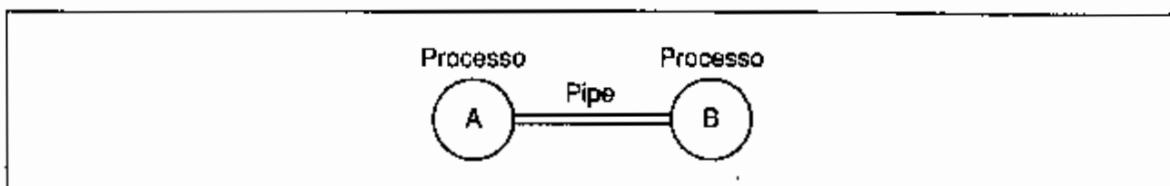


Figura 1.16 Due processi connessi da una pipe.

`mount` permette al file system sul dischetto di essere attaccato al file system radice, là dove il programma vuole che venga attaccato. In Figura 1.15(b) il file system su dischetto è stato montato nella directory `b`, permettendo l'accesso ai file `/b/x` e `/b/y`. Se la directory `b` contenesse dei file, questi non sarebbero accessibili mentre il dischetto è montato, dal momento che `/b` fa riferimento alla directory radice del dischetto. (Non essere in grado di accedere a questi file non è così grave come sembra: i file system sono quasi sempre montati in directory vuote.) Se un sistema ha più dischi rigidi, questi possono essere montati in un'unica struttura ad albero nello stesso modo.

Un altro concetto importante in UNIX è quello di **file speciale**: i file speciali vengono forniti per fare in modo che i dispositivi di ingresso/uscita sembrino file; in questo modo, è possibile leggere da o scrivere su di un dispositivo usando le stesse chiamate di sistema previste per la lettura e la scrittura dei file. Esistono due tipi di file speciali: i **file speciali a blocchi** ed i **file speciali a caratteri**. I file speciali a blocchi vengono usati per modellare dispositivi costituiti da un insieme ordinato di blocchi ad accesso casuale, come per esempio i dischi. Con l'apertura un file speciale a blocchi e la lettura di un blocco, per esempio il numero 4, un programma può accedere direttamente al quarto blocco sul dispositivo, senza tener conto della struttura del file system contenuto sul dispositivo. Analogamente, i file speciali a caratteri vengono utilizzati per modellare stampanti, moderni e gli altri dispositivi che accettano in ingresso o forniscono in uscita un flusso di caratteri. Per convenzione tutti i file speciali vengono mantenuti nella directory `/dev`; ad esempio, `/dev/lp` potrebbe essere la stampante.

L'ultima caratteristica che discuteremo in questo sommario, le **pipe**, si riferisce sia ai processi sia ai file. Una **pipe** è una specie di pseudo-file che può essere usato per collegare due processi, come illustra la Figura 1.16. Se i processi A e B vogliono comunicare utilizzando una pipe, devono inizializzarla in anticipo; quando un processo A vuole spedire dati al processo B, scrive sulla pipe come se fosse un file in uscita, e il processo B può leggere i dati leggendo dalla pipe, come se fosse un file d'ingresso. Quindi, la comunicazione fra processi in UNIX somiglia molto alle comuni letture e scritture su file; per di più, l'unico modo che un processo ha per scoprire che un file su cui sta scrivendo non è un vero file, ma una pipe, è di eseguire una chiamata di sistema speciale. I file system sono molto importanti e avremo molto da dire in merito nel Capitolo 6, e anche nei Capitoli 10 e 11.

1.5.6 Sicurezza

I calcolatori contengono grandi quantità di informazioni che spesso gli utenti vogliono mantenere riservate: tra esse, posta elettronica, strategie d'affari, dichiarazioni dei redditi e molte altre. È compito del sistema operativo amministrare la sicurezza del sistema, in modo che questi file, per esempio, siano accessibili solo dagli utenti autorizzati.

Come semplice esempio, solo per dare un'idea di come può lavorare la sicurezza, si consideri UNIX: i file in UNIX sono protetti assegnando ad ognuno di essi un codice binario a 9 bit. Il codice di protezione è costituito da tre campi di 3 bit, uno per il proprietario del file, uno per gli altri membri del gruppo a cui appartiene il proprietario, (gli uten-

ti vengono divisi in gruppi dall'amministratore di sistema) e uno per tutti gli altri; ogni campo ha un bit per l'accesso in lettura, un bit per l'accesso in scrittura, un bit per l'accesso in esecuzione: questi 3 bit sono conosciuti come i bit *rwx*. Per esempio, il codice di protezione *rwxr-x--x* significa che il proprietario può leggere (read), scrivere (write) o eseguire (execute) il file, gli altri membri del gruppo possono leggerlo o eseguirlo (ma non scriverlo) e tutti gli altri possono eseguirlo (ma non leggerlo o scriverlo). Per una directory, *x* indica i permessi di ricerca. Un trattino indica che il permesso corrispondente è assente.

In aggiunta alla protezione di file, ci sono molte altre questioni relative alla sicurezza, tra le quali proteggere il sistema da intrusioni non autorizzate, sia umane che non (ad esempio, virus). Daremo un'occhiata a vari problemi di sicurezza nel Capitolo 9.

1.5.7 La shell

Il sistema operativo è il codice che esegue le chiamate di sistema; editori, compilatori, assemblatori, linker e interpreti di comandi non fanno per niente parte del sistema operativo, anche se sono importanti e utili. A rischio di confondere in qualche modo le cose, in questa sezione ci occuperemo brevemente dell'interprete dei comandi di UNIX, chiamata *shell*, che sebbene non sia parte del sistema operativo, ne utilizza pesantemente molte caratteristiche e serve quindi come buon esempio di come possano essere usate le chiamate di sistema. La shell è anche l'interfaccia principale tra un utente seduto al suo terminale ed il sistema operativo, a meno che l'utente non stia usando un'interfaccia grafica. Esistono molte shell, tra cui *sh*, *csh*, *ksh* e *bash*, e tutte supportano le funzionalità descritte di seguito, che derivano dalla shell originale (*sh*).

Quando un utente si collega al sistema viene fatta partire una shell. La shell usa il terminale come ingresso e uscita standard, e parte scrivendo un *prompt*, un carattere come il segno del dollaro, che dice all'utente che la shell è in attesa di un comando. Se ora l'utente batte,

```
date
```

per esempio, la shell crea un processo figlio e esegue il programma *date* come figlio. Mentre il processo figlio è in esecuzione, la shell aspetta che termini; quando il figlio finisce, la shell scrive di nuovo il prompt e cerca di leggere la prossima linea in ingresso. L'utente può specificare la redirezione dell'uscita standard verso un file scrivendo, per esempio,

```
date >file
```

Lo standard input può essere rediretto in maniera analoga, come in

```
sort <file1 >file2
```

che invoca il programma *sort* con l'ingresso preso da *file1* e l'uscita spedita a *file2*.

L'uscita di un programma può essere usata come ingresso di un altro programma, connettendo i due tramite una pipe. Quindi

```
cat file1 file2 file3 | sort >/dev/lp
```

chiama il programma *cat* per concatenare tre file, e spedisce l'uscita a *sort* per mettere le righe in ordine alfabetico. L'uscita di *sort* è rediretta al file */dev/lp*, che è un nome tipico della stampante.

Se un utente mette una "&" (ampersand) dopo un comando, la shell non attende che questo venga completato, ma restituisce immediatamente il prompt. Di conseguenza

```
cat file1 file2 file3 | sort >/dev/lp &
```

inizializza l'ordinamento come processo in background (in sottofondo), permettendo all'utente di continuare a lavorare normalmente mentre l'ordinamento procede. La shell possiede molte altre caratteristiche interessanti, che non abbiamo lo spazio per discuterne qui. Molti libri su UNIX parlano della shell a diversi livelli, ad esempio, Kernighan e Pike, 1984; Kochan e Wood, 1990; Medinets, 1999; Newham e Rosenblatt, 1998 e Robbins, 1999.

1.5.8 Riciclaggio di concetti

L'informatica, come molti campi, è ampiamente guidata dalla tecnologia. La ragione per cui gli antichi romani non avevano le auto non è perché amavano così tanto camminare: è perché non sapevano come costruirle. I personal computer esistono non perché milioni di persone avessero un qualche desiderio a lungo represso di possedere un calcolatore, ma perché oggi è possibile costruirli in modo economico. Spesso dimentichiamo quanto la tecnologia influenza la nostra visione dei sistemi e questo è, di quando in quando, un punto su cui meditare.

In particolare, accade di frequente che un cambiamento nella tecnologia renda qualche idea obsoleta, e questa velocemente scompare; ma un altro cambiamento nella tecnologia potrebbe resuscitarla. Questo è particolarmente vero quando il cambiamento ha a che fare con le prestazioni relative delle diverse parti del sistema; ad esempio, quando le CPU divennero molto più veloci delle memorie, le cache divennero importanti per velocizzare la memoria "lenta". Se nuove tecnologie di memoria un giorno rendessero le memorie molto più veloci delle CPU, le cache scomparirebbero; e se una nuova tecnologia di CPU le rendesse di nuovo più veloci delle memorie, le cache ricomparirebbero. In biologia, l'estinzione è per sempre, ma in informatica è un fenomeno che talvolta dura pochi anni.

Come conseguenza di questa transitorietà, in questo libro ogni tanto daremo un'occhiata a concetti "obsoleti", cioè idee che non sono ottimali per la tecnologia corrente. Siccome dei cambiamenti nella tecnologia potrebbero reintrodurre alcuni dei così detti "concetti obsoleti", è importante capire perché un concetto è obsoleto e quali cambiamenti nell'ambiente lo possono reintrodurre.

Per chiarire questo punto, consideriamo alcuni esempi. I primi calcolatori avevano insiemi di istruzioni cablati a livello di hardware, per cui le istruzioni venivano eseguite direttamente dall'hardware e non potevano essere cambiate. Poi venne la microprogrammazione, nella quale un interprete sottostante eseguiva le istruzioni software e, di conseguenza, l'esecuzione cablata a livello di hardware divenne obsoleta. In seguito vennero inventati i calcolatori RISC, e la microprogrammazione (cioè l'esecuzione interpretata) divenne obsoleta perché l'esecuzione diretta era più veloce. Ora stiamo assistendo alla risurrezione dell'interpretazione sotto forma di applet Java, che vengono spediti su Internet e interpretati all'arrivo. La velocità di esecuzione non è sempre cruciale; perché i ritardi dovuti alla rete sono così grandi che tendono a dominare; ma anche questo potrebbe cambiare, un giorno.

I primi sistemi operativi allocavano i file su disco posizionandoli in settori contigui, uno dopo l'altro. Sebbene questo schema fosse facile da implementare, non era flessibile perché quando un file cresceva, non c'era abbastanza spazio per memorizzare il resto. Così il concetto di file allocati in modo contiguo venne scartato come obsoleto; finché non

arrivarono i CD-ROM. Qui il problema di file che crescono non esiste, ed improvvisamente, la semplicità dell'allocazione contigua dei file venne vista come una grande idea, e i file system dei CD-ROM, ora, sono basati su di essa.

Come nostra idea finale, consideriamo il linking dinamico. Il sistema MULTICS fu progettato per essere in esecuzione giorno e notte senza interruzioni; per correggere i bachi (bug) nel software, era necessario un modo per sostituire le procedure di libreria mentre venivano usate, ed il concetto di linking dinamico venne inventato per questo scopo. Dopo la morte di MULTICS, il concetto per un po' venne dimenticato, poi venne riscoperto quando i moderni sistemi operativi avevano bisogno di un modo per permettere a molti programmi di condividere le stesse procedure di libreria senza averne copie proprie (perché le librerie grafiche erano diventate troppo grandi). Ancora una volta, la maggior parte dei sistemi ora fornisce qualche forma di linking dinamico. La lista potrebbe continuare, ma questi esempi dovrebbero bastare: un'idea che è obsoleta oggi potrebbe essere la regina della festa domani.

La tecnologia non è il solo fattore che guida i sistemi e il software: anche l'economia gioca un ruolo importante. Negli anni '60 e '70, molti terminali erano terminali a stampa meccanica o CRT orientati a caratteri 25 x 80 piuttosto che terminali grafici a mappa di memoria. Questa scelta non era solo una questione di tecnologia, perché terminali grafici a mappa di memoria erano già in uso prima del 1960, solo che costavano molte decine di migliaia di dollari ciascuno. Solo quando il prezzo scese enormemente la gente (i non militari) pensarono di dedicare un terminale a ogni singolo utente.

1.6 Le chiamate di sistema

L'interfaccia tra il sistema operativo e i programmi utente è definita dall'insieme di chiamate di sistema fornite dal sistema operativo. Per capire davvero cosa fa il sistema operativo, dobbiamo esaminare quest'interfaccia da vicino: le chiamate di sistema disponibili nell'interfaccia cambiano da sistema operativo a sistema operativo (sebbene i concetti sottostanti tendano a essere simili).

Siamo così obbligati a fare una scelta tra (1) generalizzazioni imprecise ("i sistemi operativi hanno chiamate di sistema per leggere i file") e (2) qualche sistema preciso ("UNIX ha la chiamata di sistema `read` con tre parametri: uno per specificare il file, uno per dove devono essere messi i dati letti, e uno per specificare quanti byte devono essere letti").

Abbiamo scelto il secondo approccio, che è più faticoso, ma dà più elementi per capire cosa fanno realmente i sistemi operativi. Sebbene questa discussione si riferisca nello specifico a POSIX (International Standard 9945-1), da cui UNIX, System V, BSD, Linux, MINIX, ecc., la maggior parte degli altri sistemi operativi moderni ha chiamate di sistema che eseguono le stesse funzioni, anche se i dettagli sono diversi. Dal momento che i veri meccanismi per eseguire una chiamata di sistema sono fortemente dipendenti dalla macchina, e spesso devono essere espressi in codice assembler, viene fornita una procedura di libreria per rendere possibile l'esecuzione di chiamate di sistema da programmi C, e spesso anche da altri linguaggi di programmazione.

È utile tenere a mente quanto segue: ogni calcolatore con una singola CPU può eseguire un'istruzione per volta. Se un processo che sta eseguendo un programma utente in modalità utente ha bisogno di un servizio di sistema, come leggere dati da un file, deve eseguire una trap o un'istruzione che esegua una chiamata di sistema per trasferire il controllo al sistema operativo. Il sistema operativo riesce a capire cosa vuole il processo chiamante esaminando i parametri, dopodiché esegue la chiamata di sistema e restituisce il controllo all'istruzione seguente alla chiamata di sistema. In un certo senso, eseguire una

chiamata di sistema è come eseguire una chiamata di procedura speciale, solo che le chiamate di sistema entrano a far parte del kernel, mentre le chiamate a procedura no.

Per rendere più chiaro il meccanismo della chiamata di sistema, diamo una rapida occhiata alla chiamata di sistema `read`. Come detto in precedenza, ha tre parametri: il primo specifica il file da leggere, il secondo specifica un buffer (area di memoria temporanea) ed il terzo indica quanti byte leggere. Come quasi tutte le chiamate di sistema, viene chiamata dai programmi C chiamando una procedura di libreria con lo stesso nome della chiamata di sistema: `read`. Una chiamata da un programma C potrebbe essere:

```
cont = read(file, buffer, nbytes);
```

La chiamata di sistema (e la procedura di libreria) restituiscono in `cont` il numero dei byte effettivamente letti: questo valore è generalmente lo stesso di `nbytes`, ma potrebbe essere inferiore, se, per esempio, durante la lettura venisse incontrata la fine del file.

Se la chiamata di sistema non può essere completata, a causa di un parametro scorretto o di un errore su disco, `cont` viene posto uguale a -1 ed il numero dell'errore viene messo in un variabile globale, `errno`. I programmi dovrebbero sempre controllare i risultati delle loro chiamate di sistema per vedere se si sono verificati errori.

Le chiamate di sistema vengono eseguite in una serie di passi: per chiarire questo concetto, esamineremo la chiamata `read` discussa prima. Prima di richiamare la procedura di libreria `read`, che effettivamente esegue la chiamata di sistema `read`, il programma chiamante prima mette i parametri sullo stack, come mostrato nei passi 1-3 in Figura 1.17. I compilatori C e C++ mettono i parametri sullo stack in ordine inverso per ragioni storiche (così facendo, il primo parametro della `printf`, la stringa di formattazione, appare sulla cima dello stack). Il primo e terzo parametro sono richiamati per valore, mentre il secondo parametro è passato per riferimento, cioè viene passato l'indirizzo (indicato da &) del buffer e non il suo contenuto; poi viene la vera chiamata alla procedura di libreria (passo 4). Quest'istruzione è la normale isfrazione di chiamata di procedura usata per richiamare tutte le procedure.

La procedura di libreria, forse scritta in linguaggio assembler, generalmente mette il numero di chiamata di sistema in un posto noto al sistema operativo, ad esempio un registro (passo 5), dopodiché esegue un'istruzione di TRAP per passare dalla modalità utente alla modalità kernel e inizia l'esecuzione a un indirizzo fisso all'interno del kernel (passo 6). Il codice del kernel comincia ad esaminare il numero di chiamata di sistema, quindi la smista al corretto gestore della chiamata di sistema, generalmente attraverso una tabella di puntatori a gestori delle chiamate di sistema, indicizzata dal numero della chiamata (passo 7). A questo punto, viene eseguito il gestore della chiamata di sistema (passo 8); una volta che il gestore ha completato il suo lavoro, il controllo può essere restituito alla procedura di libreria nello spazio utente, all'istruzione che segue la TRAP (passo 9), dopodiché si ritorna al programma utente nel modo usuale (passo 10).

Per completare il lavoro, il programma utente deve liberare lo stack, come fa dopo ogni chiamata di procedura (passo 11). Supponendo che lo stack decresca, come fa spesso, il codice compilato aumenta lo stack pointer esattamente di quanto basta per rimuovere i parametri posti sullo stack prima della chiamata a `read`. Il programma è ora libero di fare quello che deve fare dopo.

Al passo 9 citato precedentemente, abbiamo detto "può essere restituito alla procedura di libreria nello spazio utente..." per una buona ragione, perché la chiamata di sistema può bloccare il programma chiamante, impedendogli di continuare: per esempio, se il programma chiamante sta provando a leggere dalla tastiera e non è ancora stato scritto nulla, deve essere bloccato. In questo caso, il sistema operativo si guarderà in giro per

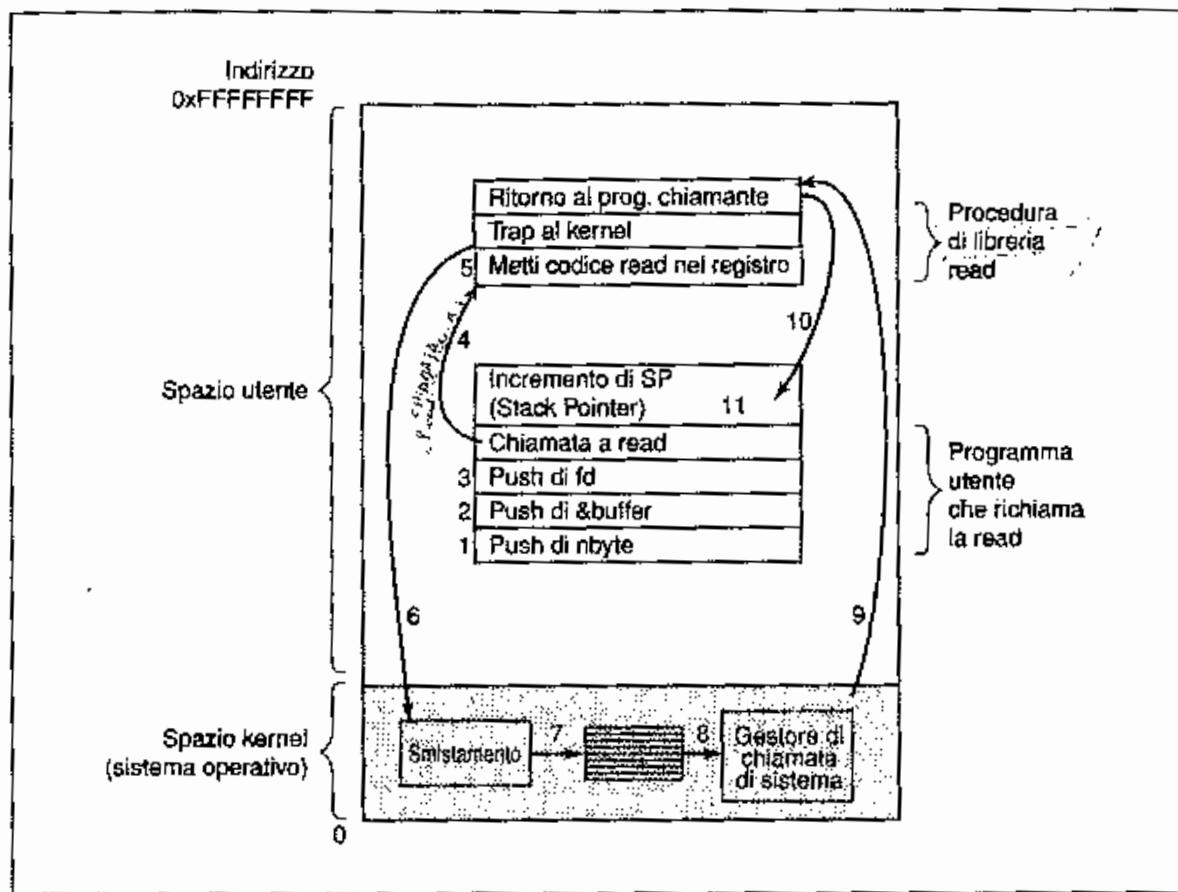


Figura 1.17 Gli 11 passi di esecuzione della chiamata di sistema `read(fd, buffer, nbyte)`

vedere se qualche altro processo può essere eseguito e, più tardi, quando l'informazione in ingresso desiderata è disponibile, questo processo riporterà l'attenzione del sistema e verranno eseguiti i passi 9-11.

Nelle sezioni che seguono, esamineremo alcune delle chiamate di sistema di POSIX usate più spesso, o, più specificatamente, le procedure di libreria che le eseguono. POSIX ha circa 100 chiamate di procedura, alcune delle più importanti sono elencate in Figura 1.18, raggruppate, per comodità, in quattro categorie. Nel testo esamineremo brevemente ciascuna chiamata per vedere cosa fa; i servizi offerti da queste chiamate determinano la maggior parte di quanto il sistema operativo deve fare, dal momento che la gestione delle risorse sui personal computer è minimale (almeno rispetto alle grandi macchine multiutente). Questi servizi comprendono la creazione e la terminazione dei processi, creare, cancellare, leggere e scrivere file, gestire directory e l'ingresso/uscita.

Come osservazione, vale la pena puntualizzare che la corrispondenza tra chiamate di procedura POSIX e chiamate di sistema non è uno-a-uno. Lo standard POSIX specifica un certo numero di procedure che un sistema conforme deve fornire, ma non specifica se siano chiamate di sistema, chiamate di libreria o qualcosa d'altro, se una procedura può essere svolta senza invocare una chiamata di sistema (per esempio senza fare trap al kernel), sarà solitamente eseguita in spazio utente per ragioni di prestazioni. Comunque, la maggior parte delle procedure POSIX invoca in effetti chiamate di sistema, spesso con una procedura che corrisponde direttamente a una chiamata di sistema, ma in alcuni casi, specialmente quando tante procedure richieste sono piccole variazioni una dell'altra, una chiamata di sistema corrisponde a più di una chiamata di libreria.

Gestione dei processi	
Chiamata	Descrizione
pid = fork()	Crea un processo figlio identico al genitore
pid = waitpid(pid, &statloc, opzioni)	Aspetta che il figlio termini
s = execve(nome, argv, ambpi)	Sostituisce l'immagine del processo
exit(stato)	Termina l'esecuzione del processo e restituisce lo stato

Gestione dei file	
Chiamata	Descrizione
fd = open(file, come, ...)	Apre un file in lettura, scrittura o entrambe
s = close(fd)	Chiude un file aperto
n = read(fd, buffer, nbyte)	Legge dati da un file e li mette in un buffer
n = write(fd, buffer, nbyte)	Scrive dati in un file, prendendoli da un buffer
posizione = lseek(fd, offset, da_dove)	Muove il puntatore a file
s = stat(name, &buf)	Restituisce le informazioni relative allo stato di un file

Gestione delle directory e del file system	
Chiamata	Descrizione
s = mkdir(name, modo)	Crea una nuova directory
s = rmdir(name)	Cancella una directory vuota
s = link(name1, name2)	Crea un nuovo riferimento, name2, che punta a name1
s = unlink(name)	Cancella un elemento di una directory
s = mount(speciale, nome, flag)	Fa il mount di un file system
s = umount(speciale)	Fa l'unmount di un file system

Miscellanea	
Chiamata	Descrizione
s = chdir(nomedir)	Cambia la directory di lavoro
s = chmod(nome, modo)	Cambia i bit di protezione di un file
s = kill(pid, segnale)	Manda un segnale a un processo
secondi = time(&secondi)	Restituisce il tempo trascorso dal 1 gennaio 1970

Figura 1.18 Alcune delle chiamate di sistema principali di POSIX. Il codice di ritorno s è -1 se si verifica un errore. I codici di ritorno sono: pid è un identificatore di processo, fd è un descrittore di file, n è un numero di byte, posizione è uno spostamento all'interno del file e secondi è il tempo trascorso.

→ I parametri sono spiegati nel testo.

1.6.1 Chiamate di sistema per la gestione di processi

Il primo gruppo di chiamate in Figura 1.18 si occupa della gestione dei processi. **Fork** è un buon punto per iniziare la discussione: tale chiamata è l'unico modo per creare un nuovo processo in UNIX. Essa crea un duplice esatto del processo originale, compresi tutti i descrittori di file e i registri. Dopo la fork, il processo originale e la copia (il padre e il figlio) procedono per strade separate; tutte le variabili hanno valori identici nel momento della fork, ma dal momento che i dati del padre sono stati copiati per creare il figlio, le modifiche successive in uno di essi non influiscono sull'altro. (Il testo del pro-

gramma, che è immutabile, è condiviso tra il padre e il figlio.) La chiamata a `fork` restituisce un valore, che è zero nel figlio e uguale all'identificatore del processo o PID (Process IDentifier) nel padre, utilizzando il PID restituito, i due processi possono capire quale è il processo padre e quale è il processo figlio.

Nella maggior parte dei casi, dopo la `fork`, il figlio ha bisogno di eseguire codice diverso rispetto al padre. Consideriamo il caso della shell: legge un comando dal terminale, si biforca creando un processo figlio, aspetta che il figlio esegua il comando e quindi legge il prossimo comando quando il figlio è terminato. Per aspettare che il figlio finisca, il padre esegue la chiamata di sistema `waitpid`, che rimane in attesa finché il figlio termina (tutti i figli, se ne esiste più di uno). `waitpid` può aspettare un figlio specifico, oppure tutti i vecchi figli impostando il primo parametro a -1; quando `waitpid` è completato, l'indirizzo puntato dal secondo parametro, `statloc`, conterrà lo stato di uscita del figlio (terminazione normale o anormale, e valore di uscita). Altre varie opzioni sono disponibili, specificate nel terzo parametro.

Consideriamo ora come `fork` è usata dalla shell: quando si digita un comando, la shell crea per biforcazione un nuovo processo, il quale deve eseguire il comando dato dall'utente, e lo fa utilizzando la chiamata di sistema `execve`, che causa la sostituzione della sua intera immagine con un file specificato nel primo parametro. (In effetti, la chiamata di sistema in sé è `exec`, ma diverse procedure di libreria la chiamano con parametri diversi e con nomi leggermente diversi; le tratteremo quindi come chiamate di sistema.) Un shell molto semplificata che illustra l'uso di `fork`, `waitpid` e `execve` è illustrata in Figura 1.19.

Nel caso più generale, `execve` ha tre parametri: il nome del file da eseguire, un puntatore a un array di argomenti, e un puntatore all'array di ambiente; descriviamo brevemente questi ultimi. Diverse routine di libreria, tra le quali `exec1`, `execv` e `execve`, permettono di specificare o tralasciare alcuni parametri in diversi modi; per tutto questo libro si utilizzerà il nome `exec` per rappresentare la chiamata di sistema invocata da tutte queste procedure di libreria.

Consideriamo il caso di un comando come il seguente

```
cp file1 file2
```

usato per copiare `file1` in `file2`; dopo che la shell si è biforcata, il processo figlio localizza ed esegue il file `cp` e gli passa i nomi del file sorgente e destinazione.

```
#define TRUE 1
while (TRUE) {
    scrivi_prompt();
    Leggi_comando(comando, parametri);
    if (fork() != 0) {           /* Ripeti per sempre */
        /* Codice del padre. */
        waitpid(-1, &stato, 0);   /* Visualizza il prompt sullo schermo */
        /* Aspetta che il figlio esca */
    } else {
        /* Codice del figlio. */
        execve(comando, parametri, 0); /* Esegui un comando */
    }
}
```

Figura 1.19 Una shell a nudo. Per tutto il libro, si suppone che TRUE sia definito come 1.

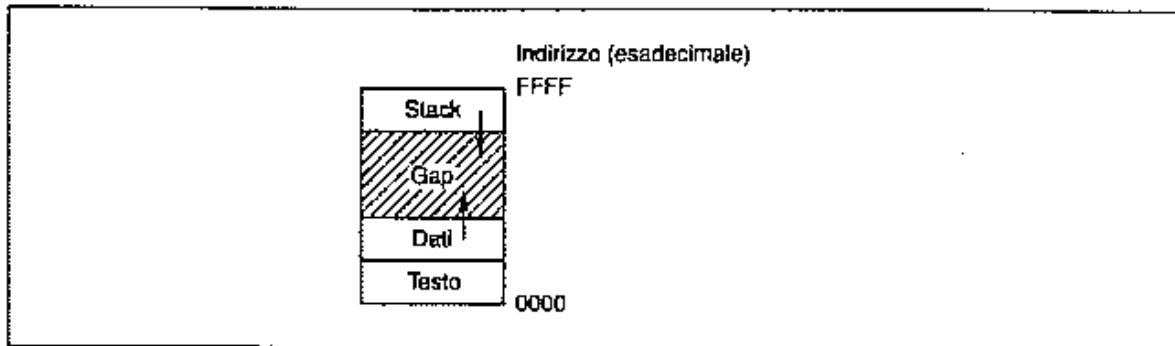


Figura 1.20 I processi hanno tre sezioni in memoria: testo, dati e stack.

Il programma principale di *cp* (e il programma principale della maggior parte degli altri programmi C) contiene la dichiarazione

```
main(argc, argv, ambp)
```

dove *argc* è il contatore del numero di elementi sulla linea di comando, compreso il nome del programma. Nell'esempio precedente, *argc* vale 3.

Il secondo parametro, *argv*, è un puntatore a un array; l'elemento *i* dell'array è il puntatore alla *i*-esima stringa sulla linea di comando. Nel nostro esempio, *argv[0]* punterebbe alla stringa "cp", *argv[1]* punterebbe alla stringa "file1" e *argv[2]* punterebbe alla stringa "file2".

Il terzo parametro di *main*, *ambp*, è un puntatore all'ambiente, un array di stringhe contenenti assegnazioni della forma *nome = valore*, usato per passare informazioni, come il tipo di terminale e il nome della home directory, a un programma. Nella Figura 1.19, nessun ambiente è passato al figlio, per cui il terzo parametro di *execve* è uno zero.

Se *exec* sembra complicata, non disperate: è (semanticamente) la più complicata fra le chiamate di sistema di POSIX, tutte le altre sono molto più semplici. Come esempio di una chiamata semplice, si consideri *exit*, che deve essere usata dai processi quando hanno finito l'esecuzione: ha un parametro, lo stato d'uscita (*exit status*, da 0 a 255), che viene restituito al processo genitore tramite il parametro *statloc* della chiamata di sistema *waitpid*.

I processi in UNIX hanno la memoria divisa in tre parti: la sezione **testo** (text segment), cioè il codice del programma, la sezione **dati** (data segment), cioè le variabili, e lo **stack** (stack segment). La sezione dei dati cresce verso l'alto e lo stack cresce verso il basso, come mostrato in Figura 1.20; tra loro c'è uno spazio d'indirizzamento inutilizzato. Lo stack cresce in questo spazio automaticamente, di quanto è necessario, mentre l'espansione della sezione dati viene eseguita esplicitamente utilizzando una chiamata di sistema, *brk*, che ne specifica il nuovo indirizzo limite. Questa chiamata, comunque, non è definita dallo standard POSIX, dal momento che i programmatore vengono incoraggiati ad utilizzare la procedura di libreria *malloc* per l'allocazione di memoria dinamica; è l'implementazione sottostante di *malloc* si pensò non fosse adatta ad essere standardizzata, dal momento che pochi programmatore la usano direttamente.

1.6.2 Chiamate di sistema per la gestione dei file

Molte chiamate di sistema sono collegate al file system; in questa sezione daremo un'occhiata alle chiamate che agiscono sui singoli file, nella prossima analizzeremo quelle che coinvolgono le directory o il file system nel suo complesso.

Per leggere o scrivere un file, questo deve essere prima aperto usando *open*: tale chia-

mata specifica il nome del file che deve essere aperto (o come nome assoluto, o relativo alla directory di lavoro), e un codice come *O_RDONLY*, *O_WRONLY* o *O_RDWR*, che sta per: apri in lettura, in scrittura o entrambe, mentre per creare un nuovo file, viene usato *O_CREAT*. Il descrittore di file restituito può essere usato per leggere o scrivere; in seguito, il file può essere chiuso da *close*, che rende disponibile il descrittore di file per essere riutilizzato da una *open* successiva.

Le chiamate più frequentemente usate sono senza dubbio *read* e *write*. Abbiamo visto prima la *read*; *write* ha gli stessi parametri.

Sebbene la maggior parte dei programmi leggano e scrivano file sequenzialmente, per alcune applicazioni è necessario accedere in modo casuale a una qualsiasi parte del file. A ciascun file è associato un puntatore, che indica la posizione corrente nel file; quando si sta leggendo (scrivendo) in modo sequenziale, generalmente punta al prossimo byte da leggere (scrivere). La chiamata *lseek* modifica il valore del puntatore di posizione, così che le successive *read* e *write* possano cominciare da un punto qualsiasi all'interno del file.

lseek ha tre parametri: il primo è il descrittore del file, il secondo è una posizione nel file, e il terzo specifica se la posizione è relativa all'inizio del file, alla posizione corrente o alla fine del file; il valore restituito da *lseek* è la posizione assoluta nel file dopo la modifica del puntatore.

Per ogni file, UNIX tiene traccia del tipo (file normale, file speciale, directory e così via), della dimensione, della data dell'ultima modifica e altre informazioni. I programmi possono richiedere di vedere queste informazioni utilizzando la chiamata di sistema *stat*. Il primo parametro specifica il file da controllare, mentre il secondo è un puntatore alla struttura dove devono essere messe le informazioni.

1.6.3 Chiamate di sistema per la gestione delle directory

In questa sezione vedremo alcune chiamate di sistema che sono relative alle directory o al file system nel suo complesso, piuttosto che a un solo specifico file, come nella sezione precedente. Le prime due chiamate, *mkdir* e *rmdir*, rispettivamente creano e rimuovono directory vuote. La chiamata successiva è *link*, il cui scopo è permettere allo stesso file di apparire sotto due o più nomi, spesso in directory diverse; un uso tipico è permettere a diversi membri dello stesso gruppo di programmazione di condividere un file comune, che compare nella directory di ciascuno di essi, anche con nomi diversi. Condividere un file non è equivalente a dare a ciascun membro del gruppo una copia privata, perché avere un file condiviso significa che le modifiche che ogni membro del gruppo fa sono immediatamente visibili agli altri membri – c'è un solo file. Quando si fanno copie di un file, le modifiche successive fatte a una copia non influenzano le altre copie.

Per vedere come lavora *link*, si consideri la situazione di Figura 1.21(a), dove ci sono due utenti, *ast* e *jim*, ciascuno con le proprie directory con alcuni file. Se *ast* esegue un programma contenente la chiamata di sistema

```
link("/usr/jim/memo", "/usr/ast/note");
```

il file *memo* nella directory di *jim* viene inserito nella directory di *ast* con il nome *note*: da questo momento, */usr/jim/memo* e */usr/ast/note* si riferiscono allo stesso file. Detto per inciso, il fatto che le directory degli utenti vengano poste in */usr*, */user*, */home* o in qualche altro posto è una decisione presa dall'amministratore di sistema.

Capire come lavora *link* probabilmente rende più chiaro cosa fa. Ogni file di UNIX ha un numero unico, il suo i-number, che lo identifica; esso è un indice nella tabella degli

/usr/ast	/usr/jim	/usr/ast	/usr/jim
16 mail 81 giochi 40 test	31 bin 70 memo 59 f.c. 38 prog1	16 mail 81 giochi 40 test 70 note	31 bin 70 memo 59 f.c. 38 prog1
(a)			(b)

Figura 1.21 (a) Due directory prima del collegamento tramite link di /usr/jim/memo alla directory di ast. (b) Le stesse directory dopo l'esecuzione di link.

i-node, uno per file, e dice a chi appartiene il file, dove sono i suoi blocchi su disco e così via. Una directory è semplicemente un file che contiene un insieme di coppie (i-number, nome ASCII); nella prima versione di UNIX, ogni elemento di directory era di 16 byte (2 byte per l'i-number e 14 byte per il nome). Ora è necessaria una struttura più complicata per supportare i nomi di file lunghi, ma concettualmente una directory è ancora un insieme di coppie (i-number, nome ASCII). In Figura 1.21, *mail* ha i-number pari a 16 e così via. Quello che fa link è semplicemente creare un nuovo elemento nella directory con un nome (forse nuovo), usando l'i-number di un file esistente; in Figura 1.21(b), due elementi hanno lo stesso i-number (70) e quindi si riferiscono allo stesso file. Se, in seguito, uno dei due viene rimosso, usando la chiamata di sistema *unlink*, l'altro rimane. Se vengono rimosso entrambi, UNIX vede che non ci sono riferimenti a quel file, (un campo nell'i-node tiene traccia del numero di elementi all'interno delle directory che si riferiscono al file), così il file viene rimosso dal disco.

Come già detto in precedenza, la chiamata di sistema *mount* permette di fondere due file system in uno; una situazione comune è quella in cui il file system radice contiene versioni binarie (eseguibili) dei comandi *comm*, e altri file usati frequentemente sul disco rigido, e l'utente può quindi inserire un dischetto con dei file da leggere nel drive del dischetto.

Eseguendo la chiamata di sistema *mount*, il file system del dischetto può essere attaccato al file system radice, come mostrato in Figura 1.22. Un'istruzione C tipica che esegue il *mount* è:

```
mount("/dev/fd0", "/mnt", 0);
```

dove il primo parametro è il nome di un file speciale a blocchi per il drive 0, il secondo parametro è il posto nell'albero dove deve essere montato, e il terzo parametro dice se

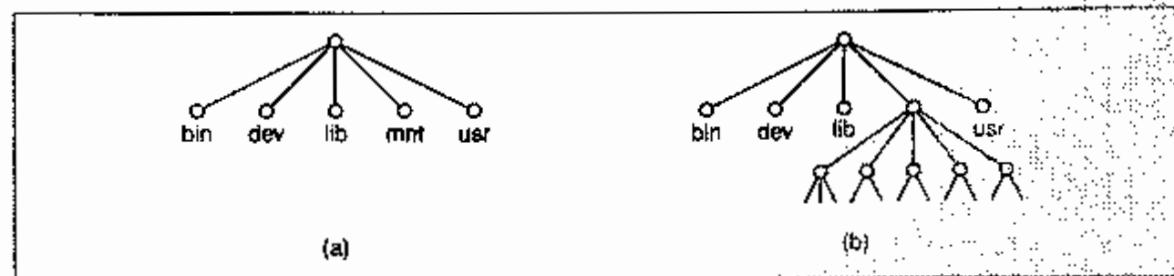


Figura 1.22 (a) File system prima dell'esecuzione di mount. (b) File system dopo l'esecuzione di mount.

il file system deve essere montato in lettura e scrittura (read-write) o solo in lettura (read-only).

Dopo la chiamata `mount`, si può accedere ad un file sul drive semplicemente usando il suo cammino dalla directory radice o dalla directory di lavoro, senza preoccuparsi del drive sul quale si trova: in realtà, è possibile montare un secondo, un terzo, un quarto drive dovunque nell'albero. La chiamata `mount` rende possibile integrare dispositivi rimovibili in una sola gerarchia di file integrata, senza doversi preoccupare di quale dispositivo contiene un file. Sebbene questo esempio riguardi i dischetti, anche i dischi fissi o porzioni di dischi fissi (spesso chiamate **partizioni** o **dispositivi secondari**), possono essere montate in questo modo; quando un file system non è più necessario, può essere smontato con la chiamata di sistema `umount`.

1.6.4 Miscellanea di chiamate di sistema

Esiste una molteplicità di altre chiamate di sistema; qui daremo un'occhiata solo a quattro di queste. La chiamata `chdir` cambia la directory di lavoro corrente. Dopo la chiamata

```
chdir("/usr/ast/test");
```

una `open` sul file `xyz` aprirà `/usr/ast/test/xyz`; il concetto di directory di lavoro elimina la necessità di scrivere cammini assoluti (lunghi) tutte le volte.

In UNIX ogni file ha una modalità utilizzata per la protezione, che comprende i bit `rwx` per il proprietario, utenti del gruppo e altri utenti. La chiamata di sistema `chmod` rende possibile cambiare la modalità di un file; ad esempio, per rendere un file accessibile solo in lettura per chiunque, eccettuato il proprietario, è possibile eseguire:

```
chmod("file", 0644);
```

La chiamata di sistema `kill` consente ad utenti ed a processi utente di inviare segnali; se un processo è preparato a catturare un segnale particolare, quando questo arriva, si esegue un gestore di segnale, mentre se non è preparato a gestire un segnale, il suo arrivo uccide il processo (da cui il nome della chiamata di sistema).

POSIX definisce diverse procedure per trattare il tempo; ad esempio, `time` semplicemente restituisce la data e l'ora correnti in secondi, con 0 che corrisponde alla mezzanotte del 1 gennaio 1970 (intesa come l'inizio del giorno, non la fine). Sui calcolatori con parole di 32 bit, il valore massimo che può essere restituito da `time` è $2^{32} - 1$ secondi (supponendo che venga usato un intero senza segno). Questo valore corrisponde a poco più di 136 anni; così, nell'anno 2106, i sistemi UNIX a 32 bit impazziranno, imitando il noto "problema Y2K" dell'anno 2000. Se attualmente avete un sistema UNIX a 32 bit, vi raccomandiamo di barattarlo con un sistema a 64 bit prima dell'anno 2106.

1.6.5 Le Win32 API di Windows

Finora ci siamo concentrati principalmente su UNIX, ma ora è tempo di dare una breve occhiata a Windows. Windows e UNIX differiscono in modo fondamentale nei rispettivi modelli di programmazione. Un programma UNIX è costituito da codice che fa qualcosa o qualcos'altro, eseguendo chiamate di sistema per fare eseguire determinati servizi, viceversa, un programma Windows è generalmente guidato dagli eventi (event driven): il programma principale aspetta che qualche evento accada, quindi chiama una procedura per gestirlo. Eventi tipici sono la battitura di un tasto, il movimento del mouse, premere un

tasto del mouse o inserire un dischetto; quindi, vengono chiamati i gestori per elaborare l'evento, aggiornare lo schermo e lo stato interno del programma. Tutto sommato, questo porta a uno stile di programmazione diverso da quello di UNIX, ma dal momento che l'interesse principale di questo libro sono la struttura e le funzioni del sistema operativo, non ci occuperemo quasi più di questi diversi modelli di programmazione.

Ovviamente, anche Windows ha delle chiamate di sistema; con UNIX, c'è una relazione quasi 1 a 1 fra le chiamate di sistema (ad esempio, `read`) e le procedure di libreria (ad esempio, `read()`) usate per invocare le chiamate di sistema. In altre parole, per ogni chiamata di sistema, c'è approssimativamente una procedura di libreria che viene richiamata per attivarla, come indicato in Figura 1.17; inoltre, POSIX ha solo un centinaio di chiamate di procedura.

Con Windows, la situazione è radicalmente diversa: tanto per cominciare, le chiamate di libreria e le vere chiamate di sistema sono totalmente spaiate; Microsoft ha definito un insieme di procedure, le **Win32 API** (**A**pplication **P**rogram **I**nterface – Interfaccia fra Applicazioni e Programmi), che si suppone che i programmati usino per ottenere servizi dal sistema operativo. Quest'interfaccia è (parzialmente) supportata da tutte le versioni di Windows, a partire da Windows 95; separando l'interfaccia dalle chiamate di sistema reali, Microsoft ha la facoltà di modificare le chiamate di sistema vere nel tempo (persino da una release all'altra), senza invalidare i programmi esistenti. Attualmente, cosa costituisce Win32 è leggermente ambiguo, dal momento che Windows 2000 ha molte chiamate nuove che non erano disponibili precedentemente; in questa sezione, Win32 indica l'interfaccia supportata da tutte le versioni di Windows.

Il numero di chiamate Win32 API è estremamente ampio, nell'ordine delle migliaia; inoltre, mentre molte di esse invocano chiamate di sistema, un considerevole numero viene eseguito nello spazio utente. Di conseguenza, con Windows è impossibile vedere cosa è una chiamata di sistema (cioè, eseguita dal kernel) e cosa è semplicemente una chiamata di libreria nello spazio utente, infatti, quella che è una chiamata di sistema in una versione di Windows può essere eseguita nello spazio utente in versioni differenti, e viceversa. Quando parliamo delle chiamate di sistema di Windows in questo libro, useremo le procedure Win32 (dove adatto) dal momento che Microsoft garantisce che queste saranno stabili nel tempo, ma vale la pena ricordare che non tutte queste sono vere chiamate di sistema (cioè trap al kernel).

Un'altra complicazione è che in UNIX la GUI (cioè X Windows e Motif) vengono completamente eseguite nello spazio utente, così le sole chiamate di sistema necessarie per scrivere sullo schermo sono `write` e poche altre chiamate secondarie. Naturalmente, ci sono molte chiamate a X Windows e alla GUI, ma queste non sono chiamate di sistema in nessun senso.

Per contrapposizione, Win32 API ha un ampio numero di chiamate per la gestione di finestre, figure geometriche, testo, caratteri, barre di scorrimento, finestre di dialogo, menu e altre caratteristiche della GUI. Nella misura in cui il sottosistema grafico viene eseguito nel kernel (vero per qualche versione di Windows, non per tutte), queste sono chiamate di sistema; a parte ciò, sono solo chiamate di libreria. Dovremmo parlare di queste chiamate in questo libro, o no? Dal momento che non sono veramente collegate alle funzioni di un sistema operativo, abbiamo deciso di no, sebbene possano essere eseguite dal kernel. I lettori interessati alle Win32 API possono consultare uno dei tanti libri sull'argomento, per esempio (Hart, 1997; Rector e Newcomer, 1997 e Simon, 1997).

Sebbene qui l'introduzione di tutte le chiamate Win32 API sia fuori questione, possiamo limitarci a quelle che approssimativamente corrispondono alle funzionalità delle chiamate di UNIX elencate in Figura 1.18, che sono elencate in Figura 1.23.

Scorriamo velocemente la lista di Figura 1.23: `CreateProcess` crea un nuovo processo, combinando il lavoro di `fork` e `execve` di UNIX, ed ha molti parametri che spe-

UNIX	Win32	Descrizione
fork	CreateProcess	Crea un nuovo processo
waitpid	WaitForSingleObject	Aspetta che un processo termini
execve	{nessuna}	CreateProcess = fork + execve
exit	ExitProcess	Termina l'esecuzione
open	CreateFile	Crea un file o apre un file esistente
close	CloseHandle	Chiude un file
read	ReadFile	Legge dati da un file
write	WriteFile	Scrive dati in un file
lseek	SetFilePointer	Muove il file pointer (riferimento alla posizione interna al file)
stat	GetFileAttributesEx	Restituisce vari attributi di file
mkdir	CreateDirectory	Crea una nuova directory
rmdir	RemoveDirectory	Rimuove una directory vuota
link	{nessuna}	Win32 non supporta i link (collegamenti)
unlink	DeleteFile	Elimina un file esistente
mount	{nessuna}	Win32 non supporta il mount
umount	{nessuna}	Win32 non supporta il mount
chdir	SetCurrentDirectory	Cambia la directory di lavoro corrente
chmod	{nessuna}	Win32 non supporta chiamate per la sicurezza (ma NT sì)
kill	{nessuna}	Win32 non supporta i segnali
time	GetLocalTime	Restituisce data e ora correnti

Figura 1.23 Le chiamate Win32 API che corrispondono approssimativamente alle chiamate UNIX di Figura 1.18.

cificano le proprietà del processo appena creato. Windows non ha una gerarchia di processi come l'ha UNIX, quindi non esiste il concetto di processo genitore e processo figlio: dopo che un processo è stato creato, creatore e creato sono alla pari. `WaitForSingleObject` viene utilizzata per aspettare un evento; è possibile attendere molti eventi. Se il parametro specifica un processo, il chiamante aspetta che il processo specificato termini, il che può essere fatto usando `ExitProcess`.

Le sei chiamate successive operano su file, e sono funzionalmente simili alle loro controparti UNIX anche se hanno parametri e dettagli diversi; grazie ad esse i file possono essere aperti, chiusi, letti e scritti, grosso modo come in UNIX. `SetFilePointer` e `GetFileAttributesEx` impostano la posizione all'interno del file, e restituiscono alcuni attributi del file.

Windows ha directory che vengono create e rimosse rispettivamente con `CreateDirectory` e `RemoveDirectory`; c'è anche una nozione di directory corrente, impostata da `SetCurrentDirectory`. La data e l'ora correnti si ottengono grazie a `GetLocalTime`.

L'interfaccia Win32 non ha collegamenti (link) ai file, non permette di montare file system, non fornisce funzionalità di sicurezza né segnali, di conseguenza le chiamate corrispondenti a quelle di UNIX non esistono. Naturalmente, Win32 ha un ampio numero di altre chiamate che UNIX non ha, in particolar modo per il trattamento della GUI, e Windows 2000 ha un sistema di sicurezza elaborato e supporta anche i collegamenti ai file.

Un ultimo appunto a proposito di Win32 è forse doveroso, per il fatto che Win32 non è molto uniforme o consistente come interfaccia; ma il principale responsabile di ciò è stata la necessità di mantenersi compatibile con le precedenti interfacce a 16 bit usate in Windows 3.x.

1.7 Struttura di un sistema operativo

Ora che abbiamo visto come un sistema operativo appare esternamente (cioè l'interfaccia verso il programmatore), è giunto il momento di dare un'occhiata all'interno: nelle sezioni che seguono, esamineremo cinque diverse strutture che sono state provate, al fine di avere una qualche idea dello spettro delle possibilità esistenti. Non abbiamo la pretesa di essere in alcun modo esaustivi, ma solo di dare un'idea di alcuni progetti effettivamente provati; i cinque progetti sono: sistemi monolitici, sistemi a livelli, macchine virtuali, exokernel e sistemi client-server.

1.7.1 Sistemi monolitici

Questo approccio, di gran lunga il più usato, può essere chiamato "Il grande pasticcio", in quanto la sua struttura consiste nell'assenza di struttura: il sistema operativo è scritto come un insieme di procedure, ciascuna delle quali può chiamarne una qualunque delle altre, ogni qualvolta ne ha bisogno. Utilizzando questa tecnica, ogni procedura nel sistema ha un'interfaccia ben definita, in termini di parametri e di risultati, e ciascuna delle procedure è libera di chiamarne un'altra, se quest'ultima fornisce alla prima una computazione di cui la prima ha bisogno.

Quando viene utilizzato questo approccio, per costruire il vero programma oggetto del sistema operativo, prima si compilano tutte le singole procedure, o i file che contengono le procedure, che in seguito vengono legate tutte insieme in un unico file oggetto tramite un linker di sistema. Non c'è essenzialmente nessun tipo di mascheramento dell'informazione: ogni procedura è visibile da ogni altra procedura (al contrario di quanto avviene in una struttura contenente moduli o package, dove la maggior parte dell'informazione è nascosta all'interno di un modulo, e dall'esterno del modulo si possono richiamare soltanto i punti di ingresso esplicitamente designati a tale scopo).

Comunque, perfino nei sistemi monolitici è possibile individuare un minimo di strutturazione: i servizi (le chiamate di sistema) forniti dal sistema operativo, vengono richiesti mettendo i parametri in luoghi ben definiti, come ad esempio sullo stack, ed eseguendo poi un'istruzione di trap. Questa istruzione fa passare la macchina dalla modalità utente alla modalità kernel, e trasferisce il controllo al sistema operativo, come illustrato al passo 6 in Figura 1.17; il sistema operativo esamina i parametri e determina quale chiamata di sistema si debba eseguire, dopodiché accede ad una tabella che contiene, nell'elemento k , un puntatore alla procedura che esegue la k -esima chiamata di sistema (passo 7 in Figura 1.17).

Questa organizzazione suggerisce una struttura di base per il sistema operativo:

1. Un programma principale che chiama le procedure di servizio richieste.
2. Un insieme di procedure di servizio che eseguono le chiamate di sistema.
3. Un insieme di procedure di utilità che aiutano le procedure di servizio.

In questo modello, per ogni chiamata di sistema c'è una procedura di servizio corrispondente che se ne occupa; le procedure di utilità svolgono compiti richiesti da diverse procedure di servizio, come, ad esempio, prelevare i dati dai programmi utente. Questa divisione delle procedure in tre livelli è illustrata in Figura 1.24.

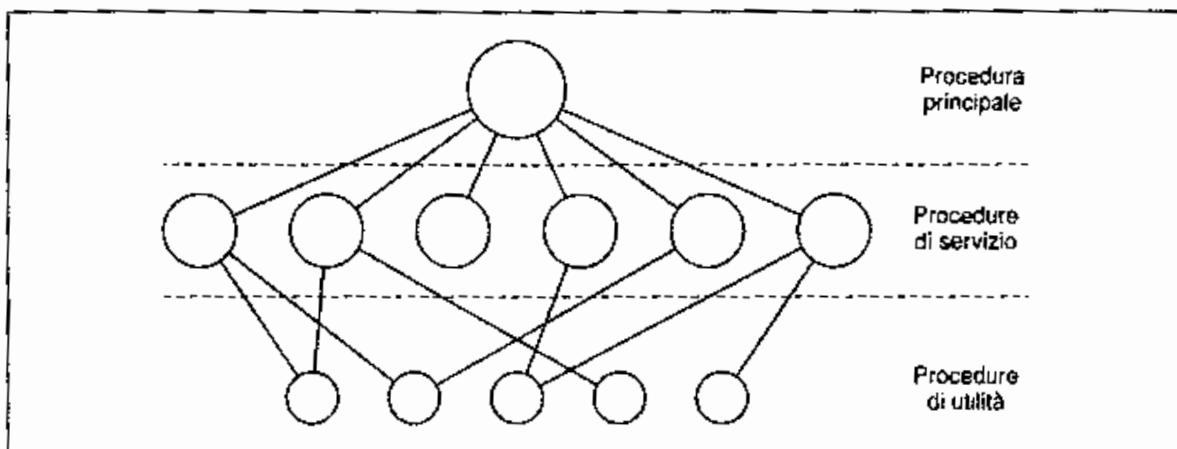


Figura 1.24 Un semplice modello di strutturazione per un sistema monolitico.

1.7.2 Sistemi a livelli

Una generalizzazione dell'approccio di Figura 1.24 consiste nell'organizzare il sistema operativo come una gerarchia di livelli, ciascuno dei quali costruito su quello sottostante. Il primo sistema operativo costruito in questo modo è stato il sistema THE realizzato alla Technische Hogeschool Eindhoven, in Olanda, da E. W. Dijkstra (1968) e dai suoi studenti. Il sistema THE era un semplice sistema batch per un calcolatore olandese, l'Electrologica X8, che aveva 32K parole da 27 bit (i bit erano costosi, in quei giorni).

Il sistema aveva 6 livelli, come illustrato in Figura 1.25; il livello 0 si occupava dell'allocatione del processore e saltava da un processo all'altro quando si verificavano interruzioni, o scadeva un timer. Sopra il livello 0, il sistema era composto da processi sequenziali, ciascuno dei quali poteva essere programmato senza dover tener conto del fatto che venivano eseguiti molti processi su un solo processore; in altre parole, il livello 0 forniva la multiprogrammazione elementare della CPU.

Il livello 1 era responsabile della gestione della memoria, ed allocava spazio per i processi in memoria principale, e su un dispositivo a tamburo (drum) di 512K parole, che veniva usato per mantenere parti di processi (pagine) per le quali non vi era spazio nella memoria principale. Sopra il livello 1, i processi non si dovevano preoccupare di dove si trovavano, se in memoria o sul tamburo, perché il livello 1 del software si faceva carico di rendere disponibili in memoria le pagine, ogni qualvolta esse risultavano necessarie.

Il livello 2 gestiva la comunicazione fra ciascuno dei processi e la console dell'operatore; al di sopra di questo livello, ogni processo aveva a disposizione una propria console. Il livello 3 si prendeva cura di gestire le unità di ingresso/uscita e di bufferizzare i flus-

Livello	Funzione
5	L'operatore
4	Programmi utente
3	Gestione dei dati in ingresso e in uscita (input/output)
2	Comunicazione fra operatore e processo
1	Gestione della memoria e del dispositivo a tamburo
0	Allocatione del processore e multiprogrammazione

Figura 1.25 Struttura del sistema operativo THE.

si di informazione da e per le unità; sopra il livello 3, ciascun processo poteva trattare con unità di ingresso/uscita astratte dotate di caratteristiche piacevoli, invece che con dispositivi fisici con molte particolarità. Al livello 4 si trovavano i programmi utente, che non si dovevano preoccupare della gestione di processi, memoria, console ed ingresso/uscita. Il processo dell'operatore di sistema era posto al livello 5.

Un'ulteriore generalizzazione del concetto di organizzazione a livelli era presente nel sistema operativo MULTICS, che, anziché a livelli, era organizzato ad anelli concentrici, con gli anelli più interni privilegiati rispetto a quelli più esterni (che è, in effetti, la stessa cosa). Quando una procedura in un anello più esterno voleva chiamarne una di un anello più interno, doveva eseguire l'equivalente di una chiamata di sistema, cioè una istruzione di TRAP, i cui parametri venivano controllati con cura per assicurarsi che fossero validi, prima di procedere con la chiamata. Sebbene in MULTICS il sistema operativo facesse parte dello spazio di indirizzamento di ciascun processo utente, l'hardware rendeva possibile che alcune procedure (in realtà, segmenti di memoria) fossero definite come protette da lettura, scrittura o esecuzione.

Sebbene nel sistema THE lo schema a livelli sia stato realizzato solo come aiuto alla progettazione, dal momento che tutte le parti del sistema venivano alla fine collegate insieme per formare un unico programma oggetto, in MULTICS il meccanismo ad anelli era presente anche a tempo di esecuzione ed era rinforzato dall'hardware. Il vantaggio del meccanismo ad anelli consisteva nel fatto che poteva essere facilmente esteso per strutturare sottosistemi utente. Per esempio, un professore poteva scrivere un programma che valutasse eassegnasse la votazione ai programmi degli studenti e mandarlo in esecuzione nell'anello n , mentre i programmi degli studenti venivano eseguiti al livello $n+1$, in maniera tale che non potessero cambiare la loro votazione.

1.7.3 Macchine virtuali

Le versioni iniziali di OS/360 erano dei sistemi strettamente a lotti (batch), tuttavia molti utenti del sistema 360 desideravano avere il timesharing, così diversi gruppi, all'interno e al di fuori di IBM, decisamente di scrivere sistemi timesharing per questo sistema operativo. Il sistema timesharing ufficiale dell'IBM, il TSS/360, fu rilasciato in ritardo e, quando finalmente uscì, era così voluminoso e lento che pochi centri decisamente di acquistarlo. Alla fine fu abbandonato, dopo che il suo sviluppo aveva richiesto circa 50 milioni di dollari (Graham, 1970), mentre un gruppo dello Scientific Center di IBM a Cambridge, nel Massachusetts, produsse un sistema radicalmente diverso che alla fine IBM accettò come prodotto e che, a tutt'oggi, è largamente utilizzato sui mainframe rimasti.

Questo sistema, chiamato originariamente CP/CMS ed ora chiamato VM/370 (Seawright e MacKinnon, 1979), era basato su un'osservazione astuta: un sistema timesharing fornisce (1) multiprogrammazione e (2) una macchina estesa con un'interfaccia più adeguata dell'hardware nudo e crudo. L'essenza del VM/370 sta nel separare completamente queste due funzioni.

Il cuore del sistema, detto monitor della macchina virtuale (virtual machine monitor), gira direttamente sull'hardware e si occupa della multiprogrammazione, fornendo al livello immediatamente superiore non una, ma parecchie macchine virtuali, come illustrato in Figura 1.26. Comunque, a differenza di tutti gli altri sistemi operativi, queste macchine virtuali non sono macchine estese, con file ed altre simpatiche caratteristiche; sono invece delle copie esatte del semplice hardware, incluse le modalità kernel/utente, l'ingresso/uscita, le interruzioni e tutto ciò che possiede una vera macchina.

Poiché ogni macchina virtuale è identica al vero hardware, ciascuna può far girare qualunque sistema operativo compatibile con l'hardware; e diverse macchine virtuali

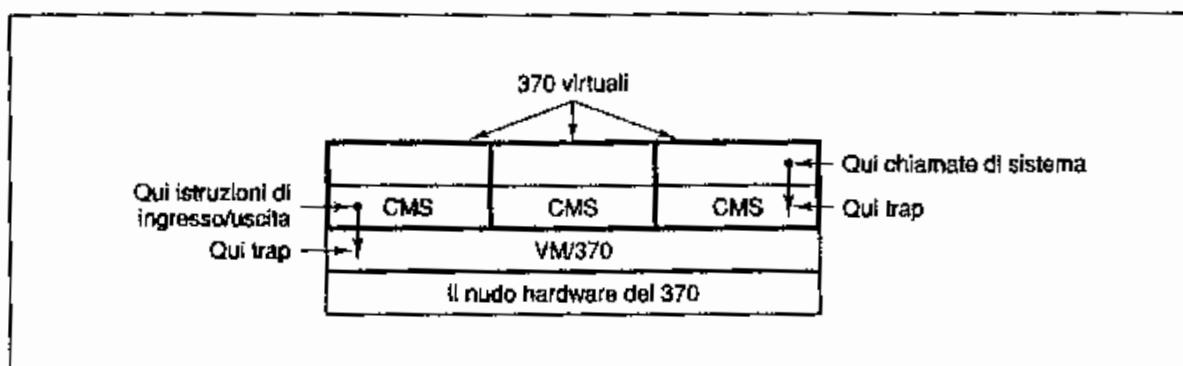


Figura 1.26 La struttura di VM/370 con CMS.

possono eseguire diversi sistemi operativi, cosa che di solito fanno: alcune eseguono uno dei discendenti di OS/360 per l'elaborazione batch, mentre altre eseguono un sistema interattivo a singolo utente chiamato CMS (Conversational Monitor System, sistema di controllo conversazionale), per utenti timesharing interattivi.

Quando un programma CMS esegue una chiamata di sistema, la chiamata viene passata al sistema operativo nella sua macchina virtuale con una trap, e non al VM/370, proprio come se il sistema operativo girasse su una macchina reale invece che su una virtuale. Il CMS poi emette le solite istruzioni hardware di ingresso/uscita per leggere il suo disco virtuale, o qualsiasi altra cosa necessaria per eseguire la chiamata; queste istruzioni di ingresso/uscita vengono catturate con una trap dal VM/370, che poi le esegue come parte della sua simulazione dell'hardware. Operando una separazione completa delle funzioni di multiprogrammazione e di presentazione di una macchina estesa, ciascuno dei due pezzi può essere molto più semplice, più flessibile e più facile da mantenere.

L'idea di una macchina virtuale oggi è usata pesantemente in un contesto differente: quando viene eseguito un vecchio programma MS-DOS su un Pentium (o su un'altra CPU a 32 bit Intel). Durante la progettazione del Pentium e del suo software, sia Intel sia Microsoft si accorsero che ci sarebbe stata una grossa richiesta di esecuzione di vecchio software sul nuovo hardware. Per questo motivo, Intel fornì sul Pentium una modalità virtuale 8086: in questo modo, la macchina si comporta come un 8086 (che è identico a un 8088 dal punto di vista del software), compreso l'indirizzamento a 16 bit con 1 MB di limite.

Questa modalità è utilizzata da Windows e da altri sistemi operativi per eseguire i programmi MS-DOS, che vengono lanciati in modalità 8086 virtuale. Finché eseguono normali istruzioni, vengono eseguiti direttamente sull'hardware, mentre quando un programma prova a fare una trap al sistema operativo per eseguire una chiamata di sistema, o prova a effettuare direttamente un'istruzione di ingresso/uscita protetta, si verifica una trap al monitor della macchina virtuale.

In tale progetto, sono possibili due varianti: nella prima, lo stesso MS-DOS viene caricato nello spazio di indirizzamento dell'8086 virtuale, così il monitor della macchina virtuale semplicemente restituisce la trap a MS-DOS, come sarebbe accaduto su un 8086 reale; quando poi MS-DOS prova a eseguire l'operazione di ingresso/uscita, questa viene catturata ed eseguita dal monitor della macchina virtuale.

Nell'altra variante, il monitor della macchina virtuale semplicemente cattura la prima trap, ed esegue l'operazione di ingresso/uscita, dal momento che conosce quali sono tutte le chiamate di sistema di MS-DOS e conosce quale trap si suppone venga eseguita. Questa variante è meno pura della prima, dal momento che ci si limita a emulare correttamente MS-DOS e nessun altro sistema operativo, come fa invece la prima. D'altro can-

to, è più veloce, dal momento che toglie la seccatura di lanciare MS-DOS per eseguire l'operazione di ingresso/uscita. Un ulteriore svantaggio dell'esecuzione reale di MS-DOS in modalità virtuale 8086 è che MS-DOS rigira parecchio i bit di abilitazione e disabilitazione delle interruzioni, che devono tutti essere emulati, con un costo considerevole.

Vale la pena di notare che nessuno di questi approcci corrisponde perfettamente a quello del VM/370, dal momento che la macchina che deve essere emulata non è un intero Pentium, ma solo un 8086. Con il sistema VM/370, è possibile eseguire lo stesso VM/370 sulla macchina virtuale, mentre con il Pentium non è possibile eseguire, ad esempio, Windows sull'8086 virtuale, perché non esiste una versione di Windows eseguibile su 8086: un 286 è il minimo persino per le più vecchie versioni, e l'emulazione del 286 non è prevista (figuriamoci l'emulazione del Pentium!). Comunque, modificando leggermente il codice di Windows, quest'emulazione è possibile, e persino disponibile in prodotti commerciali.

Un'altra area dove vengono utilizzate le macchine virtuali, ma in modo alquanto differente, è per eseguire programmi Java. Quando Sun Microsystem inventò il linguaggio di programmazione Java, inventò anche una macchina virtuale, cioè un'architettura di calcolatore, chiamata JVM (Java Virtual Machine, macchina virtuale per Java). Il compilatore Java produce codice per la JVM, che viene normalmente eseguito da un interprete JVM software; il vantaggio di questo approccio è che il codice JVM può essere inviato via Internet a un altro calcolatore che abbia un interprete JVM e ivi eseguito. Se il compilatore avesse prodotto programmi binari per SPARC o Pentium, ad esempio, questi non potrebbero essere inviati ed eseguiti dovunque così facilmente. (Naturalmente, Sun avrebbe potuto sviluppare un compilatore che produce codice binario per SPARC, e quindi distribuire un interprete per SPARC, ma JVM è un'architettura più semplice da interpretare.) Un altro vantaggio di usare JVM è che se l'interprete è implementato come si deve, il che non è completamente banale, i programmi JVM in arrivo possono essere controllati per la sicurezza, e quindi eseguiti in un ambiente protetto, in modo che non possono rubare dati o fare danni.

1.7.4 Exokernel

Con il VM/370, ogni processo utente ha una copia del vero calcolatore, mentre con la modalità 8086 virtuale sul Pentium, ogni processo utente ha una copia esatta di un calcolatore diverso da quello che sta effettivamente usando. Andando un passo avanti, i ricercatori del M.I.T. hanno costruito un sistema che dà ad ogni utente un clone del calcolatore reale, ma con un sottoinsieme di risorse (Engler e altri, 1995), in modo che una macchina virtuale possa avere a disposizione i blocchi dallo 0 al 1023, un'altra potrebbe avere dal blocco 1024 al 2047 e così via.

Al livello più basso, in esecuzione in modalità kernel, c'è un programma chiamato exokernel, il cui compito è allocare risorse alle macchine virtuali e controllarne i tentativi di utilizzo, per essere sicuri che nessuna macchina stia tentando di utilizzare le risorse di qualche altra. Ogni macchina virtuale di livello utente può eseguire il proprio sistema operativo, come accadeva sul VM/370 e sull'8086 virtuale del Pentium, tranne che ciascuna deve limitarsi ad utilizzare solo le risorse che ha richiesto e che le sono state allocate.

Il vantaggio dello schema exokernel è che risparmia un livello di mapping: negli altri progetti, ogni macchina virtuale pensa di avere il proprio disco, con i blocchi che vanno dallo 0 a un qualche massimo, per cui il monitor della macchina virtuale deve mantenere delle tabelle per convertire gli indirizzi del disco (e tutte le altre risorse). Con l'exokernel, questa conversione non è necessaria: l'exokernel ha solo bisogno di tenere traccia di

quale macchina virtuale è stata assegnata a quale risorsa. Questo metodo ha inoltre il vantaggio di separare la multiprogrammazione (nell'exokernel) dal codice del sistema operativo utente (nello spazio utente), ma con meno overhead, dal momento che tutto quello che l'exokernel deve fare è tenere le macchine virtuali fuori dagli spazi altrui.

1.7.5 Il modello client-server

Il VM/370 guadagna molto in semplicità poiché sposa una gran parte del codice tradizionale del sistema operativo (cioè quella che implementa la macchina estesa) a un livello più alto, il CMS. Tuttavia, lo stesso VM/370 è ancora un programma complesso, poiché la simulazione completa di molti 370 virtuali non è così semplice (soprattutto se lo si vuol fare con un'efficienza ragionevole).

Una tendenza dei moderni sistemi operativi è quella di sviluppare ulteriormente l'idea di spostare il codice verso livelli superiori, e rimuoverne il più possibile dal sistema operativo, lasciando un minimo microkernel. L'approccio comune implementa la maggior parte delle funzioni del sistema operativo attraverso processi utente; e per richiedere un servizio, come ad esempio la lettura di un blocco di un file, un processo utente (detto il **processo client**) spedisce la richiesta ad un **processo server**, che poi esegue il servizio e restituisce la **risposta**.

In questo modello, illustrato in Figura 1.27, il kernel si occupa solo della gestione della comunicazione tra client e server; dividendo il sistema operativo in parti, ciascuna delle quali gestisce solo un aspetto del sistema, come ad esempio la gestione dei file, quella dei processi, del terminale, o della memoria, ogni parte diventa piccola e maneggevole. Inoltre, poiché tutti i servizi vengono eseguiti come processi utente, e non in modalità

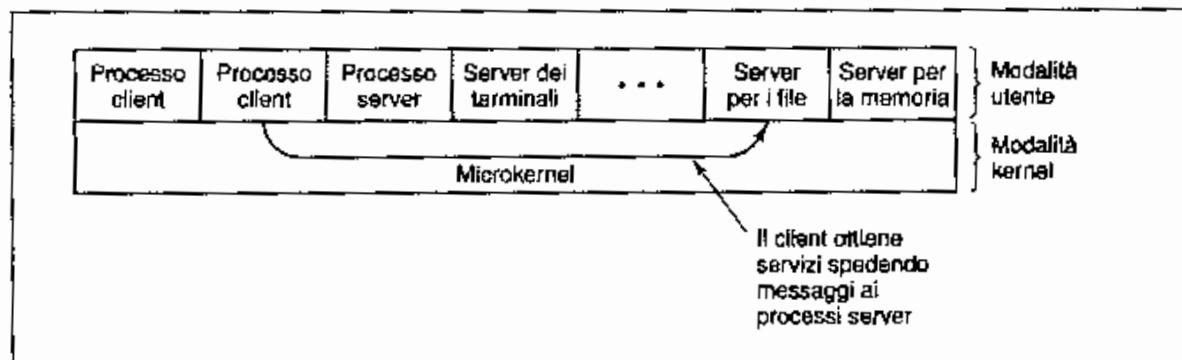


Figura 1.27 Il modello client-server.

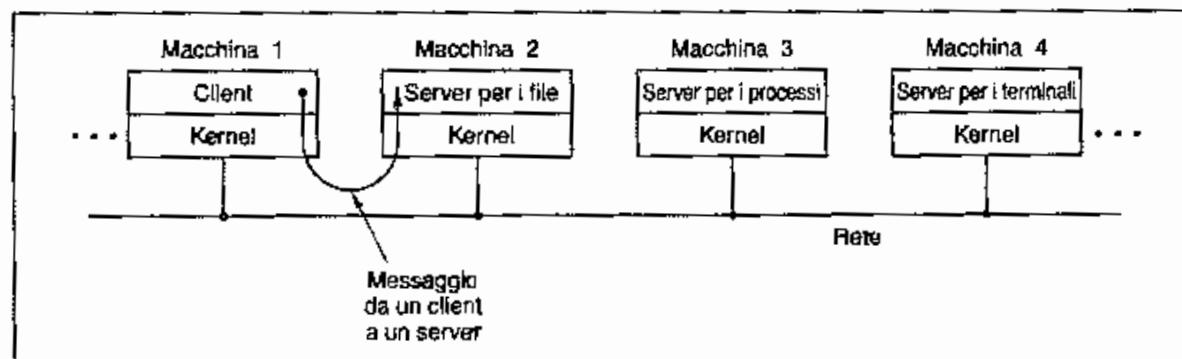


Figura 1.28 Il modello client-server in un sistema distribuito.

kernel, non si ha accesso diretto all'hardware, quindi, se ci fosse un baco nella gestione dei file, questo servizio potrebbe bloccarsi, senza causare il blocco dell'intera macchina.

Un altro vantaggio del modello client-server è la sua adattabilità ad essere utilizzato nei sistemi distribuiti (si veda la Figura 1.28): se un client comunica con un server inviandogli un messaggio, non ha bisogno di sapere se il messaggio è gestito localmente sulla sua macchina, o se invece è stato inviato attraverso una rete verso un server su una macchina remota. Per ciò che riguarda il client, nei due casi accade sempre la stessa cosa: viene spedita una richiesta, e viene ricevuta una risposta.

La figura riportata sopra, dove un kernel gestisce solo il trasferimento dei messaggi dai client ai server e viceversa, non è completamente realistica: alcune funzioni del sistema operativo (come il caricamento dei comandi nei registri dei dispositivi fisici di ingresso/uscita) sono difficili, se non impossibili, da realizzare attraverso programmi che girano nello spazio utente. Ci sono due modi per affrontare il problema: uno è quello di permettere che alcuni processi server critici (come i driver dei dispositivi di ingresso/uscita) vengano eseguiti in modalità kernel, con un accesso completo a tutto l'hardware, ma comunichino ancora con gli altri processi usando il normale meccanismo a messaggi.

L'altro modo è quello di costruire una quantità minima di meccanismi nel kernel, ma lasciare le politiche di decisione ai processi server che vengono eseguiti nello spazio utente (Levin e altri, 1975). Ad esempio, il kernel potrebbe riconoscere che un messaggio spedito a un certo indirizzo speciale significa caricare il contenuto del messaggio stesso nei registri di ingresso/uscita di un certo disco, per iniziare un'operazione di lettura. In questo caso, il kernel non controlla nemmeno i byte contenuti nel messaggio, per vedere se sono validi o significativi, ma si limita semplicemente a copiarli alla cieca nei registri di dispositivo del disco. (Ovviamente, bisogna usare un qualche meccanismo per limitare l'uso di questi messaggi solo ai processi autorizzati.) La divisione fra meccanismi e politiche è un concetto importante, che ritornerà più volte nei sistemi operativi in contesti diversi.

1.8 Ricerche sui sistemi operativi

L'informatica è un campo in rapida crescita ed è difficile prevedere dove andrà: i ricercatori universitari e quelli che lavorano per i laboratori delle industrie escogitano costantemente nuove idee, alcune delle quali non portano a nulla, ma altre diventano capisaldi di prodotti futuri e hanno un forte impatto sull'industria e sugli utenti; dire quale sarà quella riuscita è più facile con il senso di poi che nella vita di tutti i giorni. Separare il grano dal loglio è particolarmente difficile, perché spesso trascorrono 20-30 anni dall'idea all'effetto.

Ad esempio, quando il presidente Eisenhower fondò l'Advanced Research Projects Agency (ARPA) del dipartimento della difesa nel 1958, stava provando ad evitare che l'esercito escludesse la marina e l'aeronautica dal bilancio di ricerca del Pentagono, non stava cercando di inventare Internet. Ma una delle cose che ARPA fece fu sovvenzionare qualche ricerca universitaria sul concetto, all'epoca oscuro, dello smistamento di pacchetti, che portò rapidamente alla prima rete sperimentale a smistamento di pacchetti, ARPANET, che nacque nel 1969. In breve, altre reti finanziate dalle ricerche di ARPA vennero collegate ad ARPANET, e nacque Internet, che, per 20 anni, venne felicemente usata dai ricercatori accademici per spedirsi l'un l'altro e-mail. Nei primi anni '90, Tim Berners-Lee inventò, presso il laboratorio di ricerca del CERN a Ginevra, il World Wide Web per il quale Marc Andreessen, all'Università dell'Illinois, scrisse un browser grafico, ed improvvisamente, Internet si riempì di adolescenti che "chattavano", ed il presidente Eisenhower si sta probabilmente rivoltando nella tomba.

Anche la ricerca sui sistemi operativi ha portato cambiamenti drastici nei sistemi pratici. Come già detto in precedenza, i primi sistemi di calcolatori commerciali furono i sistemi batch, finché, nei primi anni '60, il M.I.T. inventò il timesharing interattivo. I calcolatori erano tutti basati su testo, finché Doug Engelbart inventò il mouse e l'interfaccia utente grafica allo Stanford Research Institute, verso la fine degli anni '60. Chi sa cosa verrà dopo?

In questa sezione e in sezioni simili del libro, daremo una breve occhiata ad alcune delle ricerche sui sistemi operativi che sono state fatte tra 5 e dieci anni fa, tanto per dare un'idea di cosa c'è all'orizzonte. Sicuramente, quest'introduzione non è esaustiva, ed è ampiamente basata su articoli pubblicati sulle più importanti riviste e alle principali conferenze, dal momento che queste idee hanno dovuto sottostare a un rigoroso e competente processo di revisione per essere pubblicate. La maggior parte degli articoli citati nelle sezioni di ricerca sono stati pubblicati da ACM, IEEE Computer Society o USENIX, e sono disponibili in rete ai membri (anche studenti) di queste organizzazioni. Per maggiori informazioni circa queste organizzazioni e le loro biblioteche elettroniche, si veda:

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Praticamente tutti coloro che effettuano ricerca sui sistemi operativi si rendono conto che i sistemi operativi di oggi sono imponenti, non flessibili, inaffidabili, insicuri e carichi di bachi, alcuni più di altri (*i nomi sono stati omessi per proteggere i colpevoli*). Di conseguenza, c'è un sacco di ricerca su come costruire sistemi flessibili e affidabili, molta della quale riguarda i sistemi con microkernel. Un kernel minimale dà una probabilità ragionevole di essere affidabile e senza errori; inoltre, questi sistemi sono flessibili, perché molte funzioni di sistema operativo vengono eseguite come processi in modalità utente, di modo che possano essere facilmente sostituite o adattate, magari anche durante l'esecuzione. Di solito, tutto quello che fa il microkernel è l'amministrazione della gestione delle risorse di basso livello, e lo smistamento di messaggi tra i processi utente.

La prima generazione di microkernel, come Amoeba (Tanenbaum e altri, 1990), Chorus (Rozier e altri, 1988), Mach (Accetta e altri, 1986) e V (Cheriton, 1988) dimostrò che questi sistemi potevano essere costruiti e fatti lavorare; la seconda generazione sta cercando di dimostrare che non solo sono in grado di lavorare, ma che lo possono fare con prestazioni elevate (Ford e altri, 1996; Hartig e altri, 1997; Liedtke 1995, 1996; Rawson 1997 e Zuberi e altri, 1999). Basandosi sui risultati pubblicati, è evidente che questo scopo è stato raggiunto.

Gran parte della ricerca sui kernel oggi è focalizzata sulla costruzione di sistemi operativi estensibili, che sono generalmente sistemi con microkernel con la capacità di estendersi o personalizzarsi in qualche direzione. Alcuni esempi sono Fluke (Ford e altri, 1997), Paramecium (Van Doorn e altri, 1995), SPIN (Bershad e altri, 1995b) e Vino (Seltzer e altri, 1996); altri ricercatori stanno cercando anche come estendere sistemi esistenti (Ghormely e altri, 1998). Molti di questi sistemi operativi permettono agli utenti di aggiungere il proprio codice al kernel, il che porta all'ovvio problema di come permettere estensioni da parte degli utenti in modo sicuro. Le tecniche comprendono l'interpretazione delle estensioni, la restrizione del codice entro sandbox, l'uso di linguaggi type-safe e la firma del codice (Grimm e Bershad, 1997; Small e Seltzer, 1998). Druschel e altri (1997) hanno presentato un punto di vista discordante, dicendo che, nei sistemi estendibili dall'utente, si è profuso uno sforzo eccessivo per la sicurezza; a loro parere, i ricercatori dovrebbero riuscire a capire quali estensioni sono utili, e devono quindi semplicemente inserirle come parte normale del kernel, senza dare agli utenti la capacità di estendere il kernel al volo.

Sebbene un modo per eliminare sistemi operativi gonfiati, pieni di errori, inaffidabili sia di renderli piccoli, un modo più radicale consiste nell'eliminare completamente il sistema operativo. Quest'approccio è stato seguito dal gruppo di Kaashoek al M.I.T., nella loro ricerca su Exokernel, nella quale l'idea è di avere un sottile strato di software che viene eseguito "sul nudo metallo", il cui unico lavoro è quello di allocare in modo sicuro le risorse hardware fra gli utenti. Ad esempio, deve decidere a chi tocca usare una parte del disco, e dove devono essere consegnati i pacchetti di rete in entrata; qualunque altra cosa è compito dei processi a livello utente, rendendo così possibile costruire sistemi operativi sia di uso generale sia altamente specializzati (Engler e Kaashoek, 1995; Engler e altri, 1995; e Kaashoek e altri, 1997).

1.9 Panoramica sul resto del libro

Ora che abbiamo completato la nostra introduzione e la nostra panoramica dei sistemi operativi, è tempo di scendere nei dettagli. Il Capitolo 2 riguarda i processi e discute le loro proprietà e come comunicano fra di loro; vengono inoltre riportati diversi esempi dettagliati di come lavora la comunicazione fra processi e di come evitare alcuni tranelli.

Il Capitolo 3 riguarda lo stallo o deadlock: abbiamo mostrato in breve cosa sono i deadlock in questo capitolo, ma c'è ancora molto da dire in proposito; in particolare, vengono discussi i modi per prevenirli o evitarli.

Nel Capitolo 4 studieremo in dettaglio la gestione della memoria; verrà esaminato l'argomento importantissimo della memoria virtuale, con i concetti collegati di paginazione e di segmentazione.

L'ingresso/uscita è trattato nel Capitolo 5; saranno trattati i concetti di indipendenza e dipendenza dai dispositivi, e verranno usati come esempio diversi dispositivi importanti, compresi dischi, tastiere e video.

Poi, nel Capitolo 6, arriviamo all'argomento cruciale del file system, che è quello che, in larga misura, l'utente vede di più; esamineremo sia l'interfaccia del file system sia la sua implementazione.

A questo punto, avremo completato il nostro studio dei principi basilari di un sistema operativo per CPU singola, ma c'è da dire di più, in particolare a proposito dei concetti avanzati. Nel Capitolo 7, esamineremo i sistemi multimediali, che hanno proprietà e requisiti diversi dai sistemi operativi convenzionali; fra le altre caratteristiche, lo scheduling e il file system sono influenzati dalla multimedialità. Un altro argomento all'avanguardia è rappresentato dai sistemi a processori multipli, compresi i multiprocessori, i calcolatori paralleli e i sistemi distribuiti, argomenti trattati nel Capitolo 8.

La sicurezza dei sistemi operativi è un argomento smisuratamente importante, che viene trattato nel Capitolo 9; fra gli argomenti discussi in questo capitolo ci sono gli attacchi, come ad esempio i virus e i worm, i meccanismi di protezione e i modelli per la sicurezza.

Infine abbiamo alcuni casi di studio di sistemi operativi reali: UNIX nel Capitolo 10 e Windows 2000 nel Capitolo 11. Il libro si conclude, nel Capitolo 12, con alcune riflessioni a proposito del progetto di sistemi operativi.

1.10 Unità di misura decimali

Per evitare confusione, vale la pena affermare esplicitamente che in questo libro, come in informatica in generale, si usano le unità di misura decimali e non le unità inglesi tra-

Notazione esponenziale	Esplicitamente	Prefisso	Notazione esponenziale	Esplicitamente	Prefisso
10^{-3}	0,001	milli	10^3	1 000	Kilo
10^{-6}	0,000001	micro	10^6	1 000 000	Mega
10^{-9}	0,000000001	nano	10^9	1 000 000 000	Giga
10^{-12}	0,000000000001	pico	10^{12}	1 000 000 000 000	Tera
10^{-15}	0,00000000000001	femto	10^{15}	1 000 000 000 000 000	Peta
10^{-18}	0,0000000000000001	atto	10^{18}	1 000 000 000 000 000 000	Exa
10^{-21}	0,000000000000000001	zepto	10^{21}	1 000 000 000 000 000 000 000	Zetta
10^{-24}	0,0000000000000000000001	yoto	10^{24}	1 000 000 000 000 000 000 000 000	Yotta

Figura 1.29 I principali prefissi del sistema metrico decimale.

dizionali (il sistema furlong-stone-fortnight). I prefissi più importanti del sistema metrico decimale sono elencati in Figura 1.29; generalmente essi sono abbreviati utilizzando le prime quattro lettere, e le unità di misura superiori all'unità (1) sono scritte con l'iniziale maiuscola. Quindi una base di dati di 1 TB occupa 10^{12} byte di memoria, e un clock di 100 psec (100 ps) scatta ogni 10^{-10} secondi. Dal momento che milli e micro cominciano entrambi con la lettera "m", deve essere effettuata una scelta: di solito, "m" sta per milli e "μ" (la lettera greca mu) sta per micro.

Vale la pena puntualizzare che per misurare la dimensione della memoria, nella normale pratica di lavoro, le unità hanno un significato leggermente diverso; qui Kilo significa 2^{10} (1024) piuttosto che 10^3 (1000), perché le memorie sono sempre potenze di 2, di conseguenza una memoria di 1 KB (KiloByte) contiene 1024 byte, e non 1000. Allo stesso modo, una memoria di 1 MB contiene 2^{20} (1 048 576) byte e una di 1 GB ne contiene 2^{30} (1 073 741 824). Al contrario, una linea di comunicazione a 1Kbps trasmette 1000 bit/s (bit al secondo) e una LAN a 10Mbps può trasmettere 10 000 000 bit/s, perché queste velocità non sono potenze di due (2). Sfortunatamente, molte persone hanno la tendenza a mescolare i due sistemi, in particolar modo per la dimensione dei dischi; per evitare ambiguità, in questo libro, useremo i simboli KB, MB e GB per indicare, rispettivamente, 2^{10} , 2^{20} e 2^{30} byte e i simboli Kbps, Mbps e Gbps per indicare, rispettivamente, 10^3 , 10^6 e 10^9 bit/s.

1.11 Sommario

I sistemi operativi possono essere osservati da due diversi punti di vista: come gestori delle risorse e come macchine estese. Secondo la visione di gestore delle risorse, il lavoro del sistema operativo è gestire in maniera efficiente le diverse componenti del sistema, mentre dal punto di vista della macchina estesa, il suo compito è quello di fornire agli utenti una macchina astratta più pratica da usare della macchina vera.

I sistemi operativi hanno una lunga storia, a partire dai giorni in cui sostituirono l'operatore, per arrivare ai sistemi multiprogrammati; i momenti salienti comprendono i primi sistemi a lotti o batch, i sistemi multiprogrammati e i personal computer.

Dal momento che i sistemi operativi interagiscono da vicino con l'hardware, qualche conoscenza dell'hardware di un calcolatore è utile alla loro comprensione: i calcolatori sono fatti di processori, memorie e dispositivi di ingresso/uscita, che sono collegati dai bus.

I concetti di base sui quali tutti i sistemi operativi sono costruiti sono processi, gestione della memoria, gestione dell'ingresso/uscita, file system e sicurezza, ognuno dei quali verrà trattato nei capitoli seguenti.

Il cuore di ogni sistema operativo è l'insieme delle chiamate che è in grado di gestire, che dicono cosa il sistema operativo è in grado di fare davvero. Per quanto riguarda UNIX, abbiamo dato un'occhiata a quattro gruppi di chiamate di sistema: il primo gruppo riguarda la creazione e la terminazione dei processi; il secondo serve a leggere e a scrivere file; il terzo è per la gestione delle directory; il quarto gruppo contiene una miscellanea di chiamate.

I sistemi operativi possono essere strutturati in diversi modi, tra i quali i più comuni sono: il sistema monolitico, una gerarchia di livelli, un sistema a macchine virtuali, un exokernel o il modello client-server.

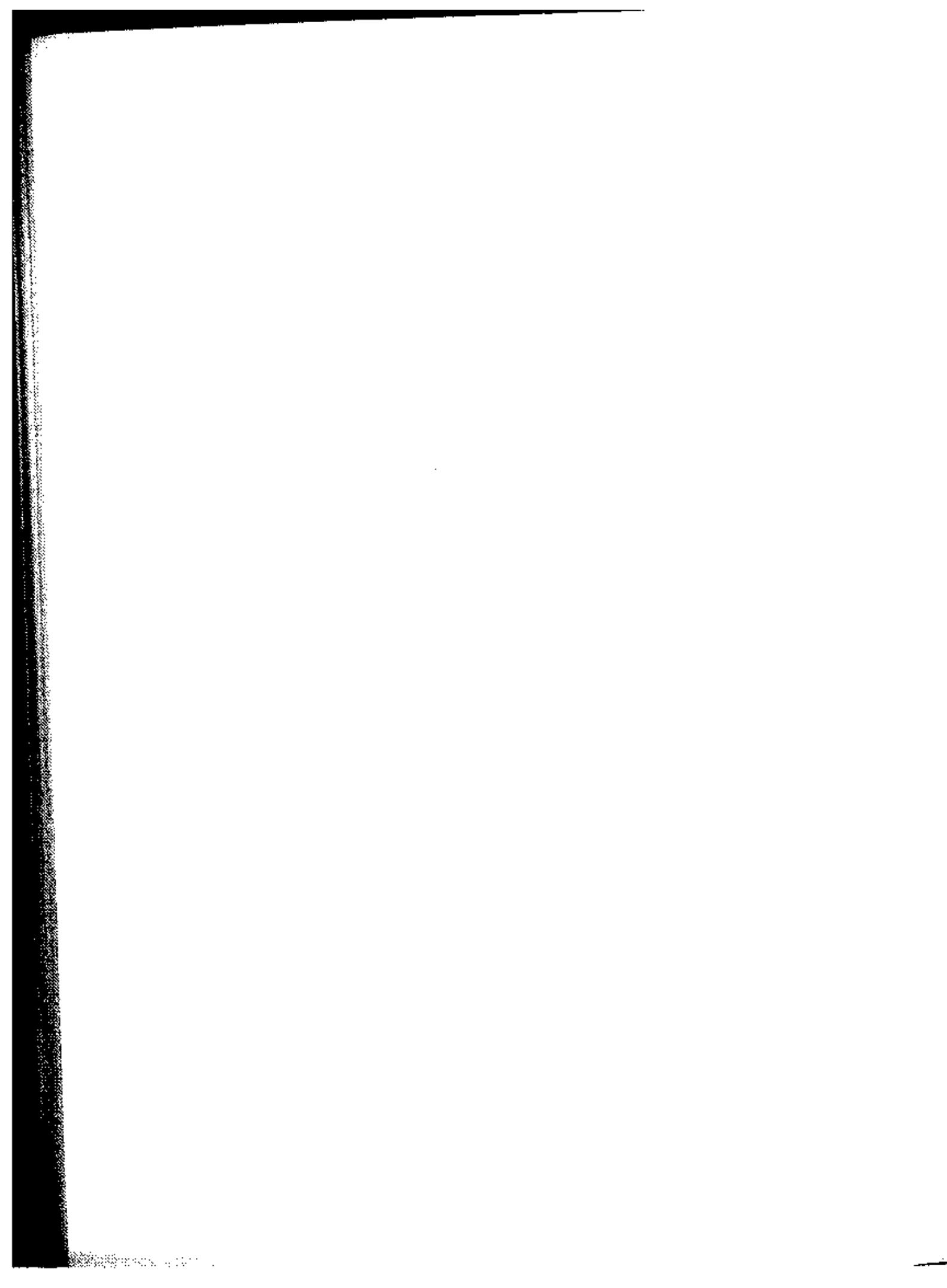
PROBLEMI

1. Quali sono le due funzioni principali di un sistema operativo?
2. Che cos'è la multiprogrammazione?
3. Che cos'è lo spooling? Pensate che in futuro i personal computer di tipo avanzato avranno lo spooling fra le loro caratteristiche di base?
4. Sui primi calcolatori, ogni byte di dati letto o scritto era gestito direttamente dalla CPU (cioè non c'era il DMA, direct memory access). Che implicazioni ha questa organizzazione sulla multiprogrammazione?
5. Perché il timesharing non era largamente diffuso fra i calcolatori della seconda generazione?
6. L'idea della famiglia di calcolatori venne introdotta negli anni '60 con i mainframe IBM System/360. Quest'idea è morta stecchita o vive ancora?
7. Una ragione per cui all'inizio le GUI furono adottate a poco a poco, fu il costo dell'hardware necessario per supportarle. Quanta memoria RAM video è necessaria per supportare una schermata di testo monocromatico a caratteri di 25 linee x 80 colonne? Quanta ne serve per una schermata di 1024 x 768 pixel, con codifica dei colori (bitmap) a 24 bit? Qual è il costo di questa RAM, utilizzando i prezzi del 1980 (\$\$/KB)? Quanto costa ora?
8. Quali delle seguenti istruzioni dovrebbero essere eseguite solo in modalità kernel?
 - (a) Disabilita tutte le interruzioni.
 - (b) Leggi l'orologio che segna l'ora e la data correnti.
 - (c) Aggiorna l'orologio che segna l'ora e la data correnti.
 - (d) Cambia la mappa di memoria.
9. Elencate alcune differenze fra i sistemi operativi dei personal computer e quelli dei mainframe.
10. Un calcolatore ha una pipeline con quattro stati. Ogni stato richiede lo stesso tempo per fare il suo lavoro, diciamo 1 ns. Quante istruzioni può eseguire, in un secondo, questa macchina?
11. Un revisore attento nota un grave errore di ortografia nel manoscritto di un libro di testo sui sistemi operativi che si accinge a stampare. Il libro ha circa 700 pagine, ciascuna di 50 righe di 80 caratteri. Quanto tempo richiederebbe la scansione elettronica del testo, nel caso della copia master posta in ciascuno dei livelli di memoria di Figura 1.7? Per il metodo di memorizzazione interna, supponiamo che il tem-

po indicato per l'accesso sia relativo a ciascun carattere, mentre per i dischi supponiamo che il tempo sia relativo a un blocco di 1024 caratteri, e per il nastro il tempo sia relativo all'inizio della procedura di accesso dei dati, mentre gli accessi successivi avvengano con la stessa velocità degli accessi a disco.

12. In Figura 1.9, l'MMU confronta l'indirizzo (virtuale) in ingresso con il registro limite, causando un errore se è troppo grande. Un progetto alternativo potrebbe essere: prima aggiungere l'indirizzo virtuale al registro base, quindi confrontare il risultato con l'indirizzo (fisico) nel registro limite. I due metodi sono logicamente equivalenti? Sono equivalenti in prestazioni?
13. Quando un programma utente effettua una chiamata di sistema per leggere o scrivere un file su disco, fornisce un'indicazione di quale file vuole, un puntatore al buffer dei dati e un numero. Il controllo viene trasferito al sistema operativo, che richiama il driver appropriato. Supponete che il driver faccia partire il disco e termini finché non si verifica una interruzione. Nel caso di lettura dal disco, ovviamente il chiamante sarà bloccato (perché non ci sono dati per lui). Nel caso di scrittura sul disco, cosa succede? È necessario bloccare il chiamante, in attesa che venga completato il trasferimento dei dati del disco?
14. Qual è la differenza fondamentale tra una trap e un'interruzione?
15. Un calcolatore usa lo schema di rilocazione di Figura 1.9(a). Un programma lungo 10000 byte viene caricato all'indirizzo 40000. Che valore hauno i registri *base* e *limite*, secondo lo schema descritto nel testo?
16. Perché si rende necessaria la tabella dei processi in un sistema timesharing? È ugualmente necessaria nel sistema operativo di un personal computer, in cui esiste un solo processo, e quel processo prende il controllo dell'intera macchina fino al suo completamento?
17. Esiste un motivo per cui potreste voler montare un file system in una directory non vuota? Se sì, qual è?
18. Per ciascuna delle chiamate di sistema che seguono, date una condizione che ne causa il fallimento: `fork`, `exec` e `unlink`.
19. Può la chiamata
`cont = write(fd, buffer, nbytes)`
 restituire in `cont` un valore diverso da quello di `nbytes`? Se sì, perché?
20. Un file il cui descrittore è `fd` contiene la seguente sequenza di byte: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Vengono eseguite le seguenti chiamate di sistema:
`lseek(fd, 3, SEEK_SET);`
`read(fd, &buffer, 4);`
 dove la chiamata `lseek` sposta il file pointer al byte 3 del file. Cosa conterrà `buffer` dopo che la `read` è stata completata?
21. Qual è la differenza essenziale fra un dispositivo speciale a blocchi ed uno a caratteri?
22. Nell'esempio in Figura 1.17, la procedura di libreria si chiama `read` e la chiamata di sistema corrispondente è `read`. È indispensabile che abbiano lo stesso nome? Se no, quale delle due è più importante?

23. Il modello client-server è diffuso nei sistemi distribuiti. Può essere usato anche in un sistema costituito da un solo calcolatore?
24. Per un programmatore, una chiamata di sistema appare come una qualunque altra chiamata a procedura di libreria. È importante che un programmatore sappia quali procedure di libreria si risolvono in chiamate di sistema? In quali circostanze e perché?
25. La Figura 1.23 mostra che un certo numero di chiamate di sistema UNIX non hanno un'equivalente nelle Win32 API. Per ognuna delle chiamate elencate per le quali non esiste l'equivalente nelle Win32 API, quali sono le difficoltà per un programmatore che voglia convertire un programma UNIX in modo che funzioni sotto Windows?
26. Ecco alcuni esercizi per impraticarsi nelle conversioni fra unità di misura: (a) Quanto è lungo un microanno, in secondi? (b) I micrometri sono spesso chiamati micron. Quanto è lungo un Gigamicron? (c) Quanti byte ci sono in una memoria di 1 TeraByte? (d) La massa della terra è 6000 Yottagrammi. A quanti Kilogrammi equivale?
27. Scrivere una shell simile a quella di Figura 1.19, ma che contenga abbastanza codice per funzionare, in modo da verificarne il funzionamento. Potete aggiungere altre caratteristiche, come la redirezione dei dati in ingresso, in uscita, le pipe e i job in background.
28. Se avete un personal computer con un sistema operativo UNIX like (Linux, MINIX, FreeBSD, eccetera), che potete tranquillamente bloccare e far ripartire, scrivete uno script di shell che cerchi di creare un numero illimitato di processi figli e osservate cosa succede. Prima di eseguire l'esperimento, dalla shell digitate sync, per fare in modo che i buffer del file system vengano scritti su disco (per evitare di rovinare il file system). NB: non provate a fare questo esperimento su un sistema condiviso da più utenti, senza prima chiedere il permesso all'amministratore di sistema: le conseguenze sarebbero immediate e molto probabilmente verreste individuati e puniti.
29. Esaminate e provate ad interpretare il contenuto di una directory UNIX like, oppure di Windows, con uno strumento come il programma UNIX od, oppure il programma MS-DOS DEBUG. Suggerimento: il modo di procedere dipende da cosa permette il sistema operativo. Un trucco potrebbe essere creare una directory su un dischetto con un sistema operativo, e leggere i dati grezzi dal disco, utilizzando un sistema operativo diverso, che permetta questo tipo di accesso.



PROCESSI E THREAD

Stiamo per imbarci nello studio dettagliato di come il sistema operativo sia progettato ed implementato. Il concetto centrale in ogni sistema operativo è quello di *processo*, che è l'astrazione per indicare un programma in esecuzione. Qualsiasi altra cosa dipende da questo concetto, ed è molto importante che il progettista del sistema operativo (nonché lo studente) comprenda appieno e il prima possibile che cos'è un processo.

2.1 Introduzione ai processi

Tutti i calcolatori moderni svolgono attività diverse nello stesso momento; mentre esegue un programma utente, un calcolatore può anche leggere dal disco e stampare su un terminale o su di una stampante. In un sistema multiprogrammato, la CPU passa da un programma all'altro, eseguendo ciascuno di essi per decine o centinaia di millisecondi; per la precisione, mentre la CPU, ad ogni singolo istante, esegue un solo programma, nell'arco di un secondo può lavorare su programmi diversi, dando all'utente l'illusione del parallelismo. Talvolta, per indicare questo continuo alternarsi fra i vari programmi da parte della CPU si parla di **pseudo-parallelismo**, per distinguerlo dal vero parallelismo hardware dei sistemi multiprocessore (che hanno due o più CPU che condividono la stessa memoria fisica). Tener traccia di molte attività parallele è difficile; di conseguenza i progettisti di sistemi operativi hanno sviluppato, nel corso degli anni, un modello concettuale (a processi sequenziali) che rende più semplice trattare con il parallelismo. Questo modello, i suoi usi e le sue conseguenze rappresentano l'argomento di questo capitolo.

2.1.1 Il modello a processi

In questo modello, tutto il software che può essere eseguito su di un calcolatore, compreso talvolta il sistema operativo, è organizzato in un certo numero di **processi sequenziali**, o, più brevemente, **processi**. Un processo non è altro che un programma in esecuzione, con i suoi valori correnti del program counter, dei registri e delle variabili; concettualmente, ogni processo dispone di una propria CPU virtuale. Naturalmente, nella realtà la CPU passa in continuazione da processo all'altro, ma per capire il sistema è molto più semplice pensare ad una collezione di processi che vengono eseguiti in (pseudo) parallelo, piuttosto che pensare ad una CPU che passa continuamente da un programma all'altro. Questo rapido cambio di contesto, avanti e indietro, viene chiamato **multiprogrammazione**, come abbiamo già visto nel Capitolo 1.

Nella Figura 2.1(a) vediamo un calcolatore che esegue in multiprogrammazione quattro programmi in memoria. In Figura 2.1(b) vediamo quattro processi, ciascuno con il proprio flusso di controllo (cioè con il proprio program counter logico), in esecuzione in maniera indipendente rispetto agli altri. Ovviamente, c'è un solo program counter fisico, e quando si esegue un processo, il suo program counter logico viene caricato nel program counter reale, mentre quando scade il tempo assegnatogli, il program counter fisico viene salvato nel program counter logico, che è posto in memoria. In Figura 2.1(c) vediamo come, su un intervallo di tempo sufficientemente lungo, ciascuno dei processi abbia fatto dei progressi, mentre, ad ogni singolo istante, solo un processo sia realmente in esecuzione.

Con la CPU che passa in continuazione da un processo all'altro, l'andamento con cui un processo svolge i suoi calcoli non risulta uniforme, e probabilmente non sarebbe nemmeno riproducibile se si rimandassero di nuovo in esecuzione gli stessi processi: di conseguenza, nella programmazione dei processi non si debbono fare ipotesi sulla temporizzazione. Consideriamo, per esempio, un processo di ingresso/uscita che mette in moto il motore di un dispositivo a nastro magnetico, esegue un ciclo a vuoto 10000 volte per attendere che il nastro acquisti velocità, ed infine lancia un comando per leggere il primo record. Se la CPU decidesse di commutare su un altro processo durante il ciclo a vuoto, il processo del nastro potrebbe ritornare in esecuzione dopo che sotto la testina di lettura è già passato il primo record. Quando un processo ha dei requisiti real time così critici, cioè quando particolari eventi devono assolutamente accadere entro un numero specificato di millisecondi, bisogna adottare misure opportune per far in modo che essi accadano veramente. Generalmente, comunque, la maggior parte dei processi non subisce conseguenze dalla multiprogrammazione della CPU o dalla diversa velocità relativa dei vari processi.

La differenza tra un processo e un programma è sottile, ma cruciale; un'analogia può

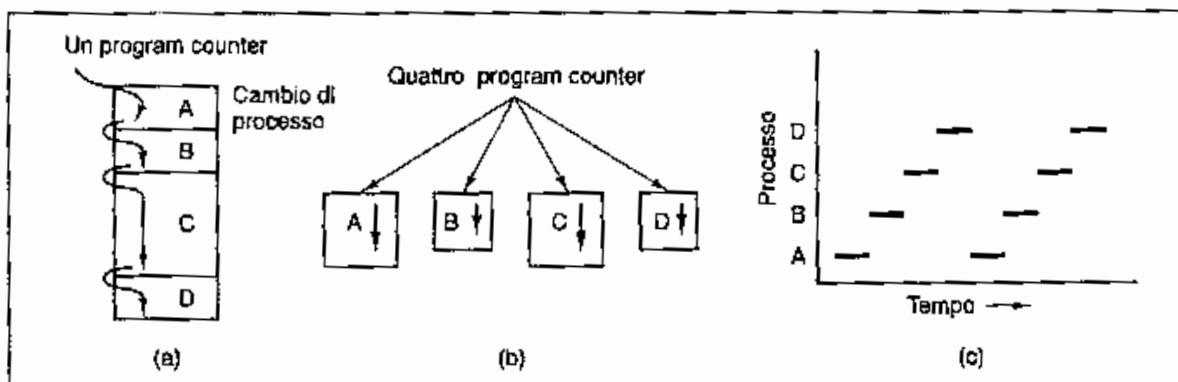


Figura 2.1 (a) Multiprogrammazione di quattro programmi. (b) Modello concettuale di quattro processi sequenziali indipendenti. (c) Solo un programma è attivo ad ogni istante.

aiutare a chiarire questa differenza. Consideriamo un informatico appassionato di cucina, che sta preparando la torta di compleanno per sua figlia. L'informatico ha la ricetta per la torta, e una cucina ben fornita con gli ingredienti necessari: farina, uova, zucchero, vaniglia e così via. In questa analogia, la ricetta è il programma (cioè l'algoritmo espresso con una notazione adeguata), l'informatico è il processore (CPU) e gli ingredienti della torta sono i dati in ingresso. Il processo è l'attività che il nostro cuoco svolge leggendo la ricetta, raccogliendo gli ingredienti, impastando e cuocendo la torta.

Ora immaginiamo che il figlio dell'informatico irrompa nella cucina piangendo e dicendo di essere stato punto da un'ape. L'informatico segna il punto della ricetta a cui è arrivato (lo stato del processo corrente viene salvato), tira fuori un libro di pronto soccorso e comincia a seguire le sue indicazioni. Questo è un esempio di come il processore cambi contesto da un processo (cucinare) ad un altro processo a priorità più alta (somministrare cure mediche), ciascuno con un programma diverso (la ricetta e il libro di pronto soccorso). Quando la puntura dell'ape è stata medicata, l'informatico ritorna alla sua torta, continuando dal punto dove era stato interrotto.

L'idea chiave, in questo caso, è che un processo è un'attività di qualche tipo, che possiede un programma, dati in ingresso, dati in uscita e uno stato. Un unico processore può essere condiviso tra parecchi processi, utilizzando un algoritmo di schedulazione per determinare quando smettere di lavorare su un processo per servirne un altro.

2.1.2 La creazione dei processi

I sistemi operativi hanno bisogno di un qualche meccanismo per essere sicuri che esistano tutti i processi che servono. In sistemi molto semplici, o in sistemi progettati per eseguire solo un'applicazione (ad esempio, il controller di un forno a microonde), all'avvio del sistema si possono avere già presenti tutti i processi che potranno essere necessari. Nei sistemi di uso generale, tuttavia, è necessario avere un qualche meccanismo per creare e distruggere processi nel corso delle operazioni. Daremo ora un'occhiata ad alcune di queste questioni.

A provocare la creazione di un processo sono, principalmente, quattro eventi:

1. L'inizializzazione del sistema.
2. L'esecuzione di una chiamata di sistema per la creazione di un processo, effettuata da un processo in esecuzione.
3. Una richiesta, da parte dell'utente, affinché venga creato un nuovo processo.
4. L'inizio di un job batch.

Quando un sistema operativo viene lanciato, vengono generalmente creati diversi processi, alcuni dei quali sono processi che vengono eseguiti in primo piano (foreground process), cioè sono processi che interagiscono con utenti (umani) e eseguono del lavoro per loro, mentre altri sono processi che vengono eseguiti in background (sullo sfondo), che non sono associati a particolari utenti, ma hanno invece qualche funzione specifica. Ad esempio, un processo in background può essere progettato per ricevere le e-mail in arrivo, dormendo per la maggior parte della giornata e scattando improvvisamente quando arrivano e-mail. Un altro processo in background potrebbe essere incaricato di accettare le richieste in arrivo di pagine Web presenti su quella macchina, risvegliandosi per soddisfare le richieste quando queste arrivano. I processi che stanno in background per gestire qualche attività come le e-mail, le pagine Web, le news, la stampa e così via vengono chiamati **demoni** (daemon). I sistemi grandi, di solito, ne hanno a dozzine: in

UNIX, si può usare il programma *ps* per avere l'elenco dei processi in esecuzione, mentre in Windows 95/98/Me, premendo CTRL-ALT-CANC, compare una finestra che mostra cos'è in esecuzione in quel momento; in Windows 2000 può essere utilizzato per lo stesso scopo il task manager.

Oltre ai processi creati a tempo di avvio, in seguito se ne possono creare di nuovi. Spesso un processo in esecuzione originerà chiamate di sistema per creare uno o più nuovi processi per aiutarlo a compiere il suo lavoro. La creazione di nuovi processi è particolarmente utile quando il lavoro da fare può essere facilmente espresso in termini di processi diversi e collegati, ma comunque indipendenti, che interagiscono tra loro. Ad esempio, se si deve prelevare una grossa quantità di dati dalla rete, per elaborarla successivamente, potrebbe convenire creare un processo che prende i dati e li mette in un buffer condiviso, mentre un secondo processo li rimuove dal buffer e li elabora. Su un multiprocessore, permettendo ad ogni processo di essere in esecuzione su CPU differenti, il job verrebbe eseguito più velocemente.

In sistemi interattivi, gli utenti possono lanciare un programma scrivendo un comando o facendo (doppio) click su un'icona, ciascuna di queste azioni provoca la creazione di un nuovo processo che esegue il programma selezionato. Nei sistemi UNIX basati sui comandi che eseguono X Window, il nuovo processo prende possesso della finestra nella quale è stato lanciato. In Microsoft Windows, quando un processo viene lanciato, non deve avere una finestra, ma può crearne una (o più), e la maggior parte lo fa. In entrambi i sistemi, gli utenti possono avere più finestre aperte contemporaneamente, ciascuna che esegue qualche processo. Usando il mouse, l'utente può selezionare una finestra e interagire con i processi, ad esempio, fornendo i dati in ingresso quando necessario.

L'ultima situazione nella quale vengono creati processi riguarda solo i sistemi batch che si trovano sui grossi mainframe, dove gli utenti possono inviare al sistema job batch (anche in remoto). Quando il sistema operativo stabilisce di avere le risorse per eseguire un altro job, crea un nuovo processo e esegue il job successivo dalla coda in ingresso.

Tecnicamente, in tutti questi casi, viene creato un nuovo processo tramite l'esecuzione di una chiamata di sistema apposita, effettuata da un processo esistente, che può essere un processo utente in esecuzione, un processo di sistema lanciato dalla tastiera o dal mouse, o un processo che gestisce l'esecuzione batch. Quello che questo processo fa è eseguire una chiamata di sistema, che dice al sistema operativo di creare un nuovo processo, e indica, direttamente o indirettamente, quale programma deve essere eseguito, in associazione ad esso.

In UNIX, esiste una sola chiamata di sistema per creare un nuovo processo: *fork*, che crea una copia esatta del processo chiamante. Dopo la *fork*, i due processi, il genitore e il figlio, hanno la stessa immagine di memoria, le stesse stringhe di ambiente, gli stessi file aperti, e questo è tutto. Normalmente, il processo figlio esegue poi una *execve*, o una chiamata di sistema simile, per cambiare la sua immagine di memoria e eseguire un nuovo programma. Ad esempio, quando un utente batte un comando nella shell, diciamo *sort*, la shell esegue una *fork*, creando un processo figlio, che esegue *sort*. La ragione di questo procedimento in due passi è di permettere al processo figlio di manipolare i propri descrittori di file, dopo la *fork*, ma prima della *execve*, per realizzare la redirezione dello standard input, dello standard output e dello standard error.

Diversamente, in Windows, una sola chiamata di funzione Win32 API, *CreateProcess*, gestisce sia la creazione del processo sia il caricamento del programma corretto nel nuovo processo. Questa chiamata ha 10 parametri, tra cui il programma da eseguire, i parametri della riga di comando da dare in pasto al programma, vari attributi di sicurezza, bit che controllano se i file aperti vengono ereditati, informazioni di priorità, una descrizione della finestra da creare per il processo (se serve) e un puntatore a una struttura nella quale informazioni relative al processo appena creato vengono restituite al

chiamante. In aggiunta alla `CreateProcess`, Win32 ha circa altre 100 funzioni per gestire e sincronizzare i processi e argomenti correlati.

Sia in UNIX sia in Windows, dopo che un processo è stato creato, il genitore ed il figlio hanno il proprio distinto spazio di indirizzamento: se uno dei due modifica una parola nel suo spazio di indirizzamento, la modifica non è visibile all'altro processo. In UNIX, lo spazio di indirizzamento iniziale del processo figlio è una copia di quello del genitore, ma ci sono due distinti spazi di indirizzamento coinvolti; nessuna memoria scrivibile è condivisa (qualche implementazione di UNIX condivide il testo del programma tra i due, dal momento che non può essere modificato). È comunque possibile che un processo appena creato condivida alcune altre risorse del proprio creatore, come i file aperti. In Windows, lo spazio di indirizzamento dei due processi è diverso dall'inizio.

2.1.3 La terminazione dei processi

Dopo che un processo è stato creato, comincia la sua esecuzione ed esegue il compito che gli è stato assegnato, qualunque esso sia. Comunque, niente dura per sempre, nemmeno i processi: presto o tardi il nuovo processo terminerà, normalmente a causa del verificarsi di una delle condizioni seguenti:

1. Terminazione normale (volontaria).
2. Terminazione con errore (volontaria).
3. Si è verificato un errore fatale (involontaria).
4. È stato ucciso (in gergo, kill) da un altro processo (involontaria).

La maggior parte dei processi termina perché ha finito il proprio lavoro. Quando un compilatore ha compilato il programma che gli è stato dato, esegue una chiamata di sistema per dire al sistema operativo che ha finito: questa chiamata è `exit` in UNIX e `ExitProcess` in Windows; anche i programmi orientati allo schermo (screen-oriented program) forniscono la terminazione volontaria. Gli elaboratori di testi, i browser per Internet e programmi simili hanno sempre un'icona o una voce di menu che l'utente può cliccare per dire al processo di rimuovere i file temporanei che ha aperto e quindi terminare.

La seconda causa di terminazione si verifica quando il processo scopre un errore fatale. Ad esempio, se l'utente scrive il comando

```
cc foo.c
```

per compilare il file `foo.c` e non esiste un file con questo nome, il compilatore, semplicemente, termina. I processi interattivi orientati allo schermo, generalmente, non terminano quando gli vengono passati parametri sbagliati; visualizzano invece una finestra di dialogo (pop up), chiedendo all'utente di riprovare.

La terza causa di terminazione è un errore causato dal processo, spesso dovuto a un errore di programmazione (bug, letteralmente scarafaggio, in gergo baco); ad esempio, l'esecuzione di istruzioni illegali, riferimenti a parti di memoria che non esistono, la divisione per 0. In alcuni sistemi (ad esempio UNIX), un processo può dire al sistema operativo che vuole gestire lui stesso certi errori, nel qual caso, quando si verifica un errore, al processo viene inviato un segnale (viene interrotto), anziché forzarne la terminazione.

Il quarto motivo per cui un processo può terminare si verifica quando un processo esegue una chiamata di sistema dicendo al sistema operativo di uccidere qualche altro processo. In UNIX questa chiamata è `KILL`, mentre la funzione Win32 corrispondente è `TerminateProcess`; in entrambi i casi, il processo killer deve avere l'autorità necessaria. In

alcuni sistemi, quando un processo termina, volontariamente oppure no, tutti i processi che ha creato vengono uccisi, ma né UNIX né Windows lavorano in questo modo.

2.1.4 Gerarchie di processi

In alcuni sistemi, quando un processo crea un altro processo, il processo genitore (detto anche padre) e il figlio continuano, in un certo senso, ad essere associati; il processo figlio può anch'esso creare altri processi, formando una gerarchia di processi. Si noti che, diversamente dalle piante e dagli animali che usano la riproduzione sessuale, un processo ha un solo genitore (ma zero, uno, due o più figli).

In UNIX, un processo e tutti i suoi figli e gli ulteriori discendenti, insieme, formano un gruppo di processi. Quando un utente invia un segnale dalla tastiera, questo è consegnato a tutti i membri del gruppo di processi correntemente associato alla tastiera (normalmente, tutti i processi attivi che sono stati creati nella finestra corrente). Individualmente, ogni processo può catturare il segnale, ignorarlo o eseguire l'azione di default, cioè essere ucciso dal segnale.

Come altro esempio di situazione in cui hanno un ruolo le gerarchie di processi, diamo un'occhiata a come UNIX si inizializza quando viene lanciato. Un processo speciale, chiamato *init*, è presente nell'immagine di avvio. Quando inizia la sua esecuzione, legge un file che dice quanti terminali ci sono, quindi si biforca creando un nuovo processo per terminale e questi processi aspettano che qualcuno esegua il login. Se il login viene eseguito con successo, il processo di login esegue una shell per accettare i comandi, che possono lanciare altri processi e così via. Così, tutti i processi nell'intero sistema appartengono a un solo albero, che ha *init* come radice.

Viceversa, Windows non ha un concetto di gerarchia di processi: tutti i processi sono uguali. Il solo caso in cui esiste qualcosa di simile a una gerarchia di processi si verifica quando un processo viene creato, perché il genitore riceve un gettone (token) speciale, chiamato **handle** o **gestore**, che può usare per controllare il figlio, ed il genitore può passare il token a qualche altro processo, invalidando la gerarchia. I processi, in UNIX, non possono diseredare i figli.

2.1.5 Gli stati dei processi

Sebbene ogni processo sia un'entità indipendente, con un proprio contatore di programma e un proprio stato interno, i processi spesso devono interagire tra di loro, ad esempio, un processo può generare dati in uscita che un altro processo usa come ingresso. Nel comando di shell:

```
cat cap1 cap2 cap3 | grep tree
```

il primo processo, che esegue *cat*, concatena tre file e li scrive in uscita; il secondo processo, che esegue *grep*, seleziona tutte le righe che contengono la parola "tree". A seconda della velocità relativa dei due processi (che dipende sia dalla relativa complessità dei programmi sia dal tempo di CPU assegnato ad ognuno dei due processi) può accadere che *grep* sia pronto all'esecuzione, ma che non ci sia alcun dato in ingresso che lo stia aspettando, quindi deve bloccarsi fino a quando non siano disponibili dei dati in ingresso.

Quando un processo si blocca, lo fa perché, da un punto di vista logico, non può continuare l'esecuzione, tipicamente perché sta aspettando qualche dato in ingresso che non è ancora disponibile. È anche possibile che un processo concettualmente pronto ed in gra-

do di cominciare l'esecuzione venga bloccato perché il sistema operativo ha deciso di allocare per un certo periodo di tempo la CPU ad un altro processo. Queste due condizioni sono completamente diverse fra loro. Nel primo caso, la sospensione è inerente al problema (non si può eseguire un comando dato dall'utente finché la riga non è stata scritta); nel secondo caso è conseguenza dei tecnicismi del sistema (non ci sono abbastanza CPU per poter assegnare ad ogni processo un processore). Nella Figura 2.2 vediamo un diagramma di stato che mostra i tre stati in cui si può trovare un processo:

1. In esecuzione (running, cioè sta veramente usando la CPU in quell'istante).
2. Pronto (ready, può essere messo in esecuzione; è temporaneamente sospeso per permettere l'esecuzione di un altro processo).
3. Bloccato (blocked, impossibilitato ad essere messo in esecuzione in attesa di un qualche evento esterno).

Dal punto di vista logico, i primi due stati sono simili: in tutti e due i casi, il processo vuole essere eseguito, solo che, nel secondo caso, non c'è, in quel momento, una CPU disponibile. Il terzo stato è diverso dai primi due, in quanto il processo non può essere eseguito, anche se la CPU non ha nient'altro da fare.

Fra questi tre stati sono possibili quattro transizioni, come mostrato in figura. La transizione 1 si verifica quando un processo scopre di non poter continuare: in alcuni sistemi il processo deve eseguire una chiamata di sistema, come `block` o `pause`, per entrare nello stato di bloccato; in altri sistemi, compreso UNIX, quando un processo legge da una pipe o da un file speciale (per esempio, un terminale) e non c'è alcun dato in ingresso disponibile, viene bloccato automaticamente.

Le transizioni 2 e 3 vengono causate dallo scheduler (schedulatore, l'ordinatore dei processi, una parte del sistema operativo), senza che i processi se ne accorgano. La transizione 2 avviene quando lo scheduler decide che il processo in esecuzione è stato in esecuzione per un tempo abbastanza lungo ed è giunto il momento di assegnare un po' del tempo di CPU ad un altro processo, mentre la transizione 3 si verifica quando tutti gli altri processi hanno avuto la loro parte ed è arrivato il momento di far ripartire il primo processo. L'attività della schedulatore, cioè decidere quale processo deve essere mandato in esecuzione e per quanto tempo, è importante e la analizzeremo più tardi, in questo stesso capitolo. Per cercare di bilanciare le due richieste in antagonismo fra loro, cioè l'efficienza del sistema e l'equità nel trattamento dei singoli processi, sono stati studiati molti algoritmi. Studieremo alcuni di essi più avanti, in questo capitolo.

La transizione 4 si verifica quando accade un evento esterno (come l'arrivo di un dato

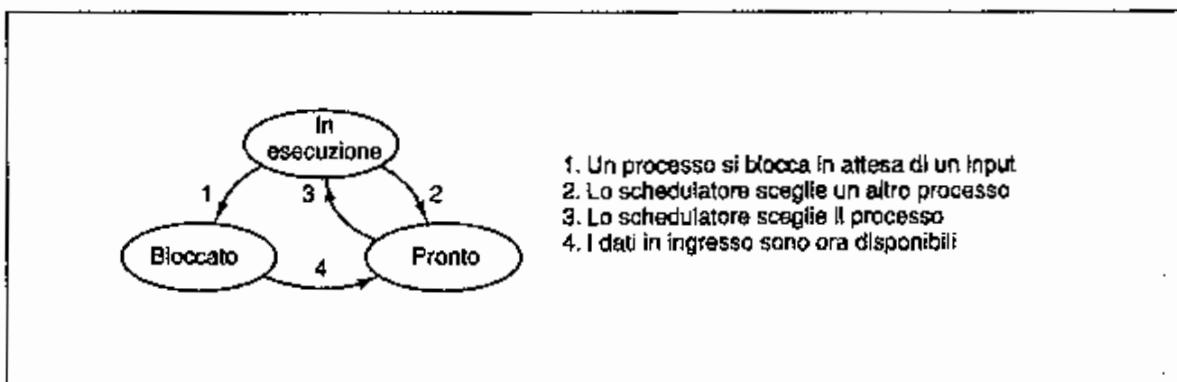


Figura 2.2 Un processo può essere in stato di esecuzione, di pronto o bloccato. Le transizioni fra questi tre stati sono quelle disegnate.

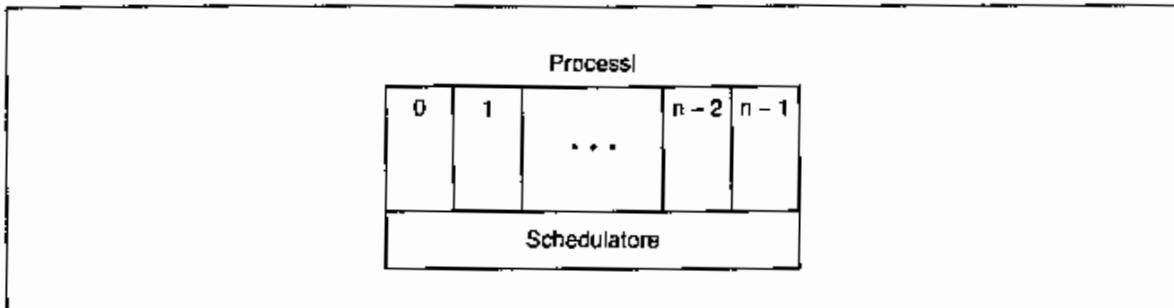


Figura 2.3 Il livello più basso di un sistema operativo strutturato a processi gestisce le interruzioni e la schedulazione. Il resto del sistema consiste di processi sequenziali.

in ingresso) che un processo attendeva; se nessun altro processo è in esecuzione in quell'istante, scatterà immediatamente una transizione di tipo 3 e il processo verrà rimesso in esecuzione, altrimenti occorrerà che il processo aspetti, in stato di *pronto* per un breve momento, finché la CPU non è disponibile e non arriva il suo turno.

Utilizzando il modello a processi, diventa molto più semplice pensare a ciò che sta accadendo dentro il sistema. Alcuni processi eseguono programmi che mettono in atto comandi battuti da un utente; altri processi sono parte del sistema e gestiscono compiti come l'esecuzione di servizi per i file, o la gestione dei dettagli per l'accesso ad un disco o ad un nastro. Quando arriva un'interruzione da disco, il sistema decide di fermare il processo corrente e di mandare in esecuzione il processo del disco, che era bloccato in attesa di quell'interruzione. Quindi, invece di ragionare in termini di interruzioni, possiamo farlo in termini di processi utente, processi dei dischi, dei terminali eccetera, che si bloccano in attesa di un qualche evento. Quando il blocco del disco è stato letto o il carattere battuto, il processo che lo stava aspettando viene sbloccato e viene ricandidato per l'esecuzione.

Questo approccio dà origine al modello mostrato in Figura 2.3. Qui, il livello più basso del sistema operativo è lo schedulatore, con un certo numero di processi che gli stanno sopra. Tutto il trattamento delle interruzioni e i dettagli sull'effettiva attivazione e blocco dei processi sono mascherati dentro lo schedulatore, che di fatto è piuttosto piccolo. Il resto del sistema operativo è piacevolmente strutturato a processi. Comunque, pochi sistemi reali hanno una struttura così carina come questa.

2.1.6 Implementazione dei processi

Per implementare il modello a processi, il sistema operativo mantiene una tabella (un vettore di strutture), chiamata la **tabella dei processi** (*process table*), con un elemento (entry) per ogni processo (alcuni autori chiamano questi elementi **blocchi di controllo dei processi**, **process control block**), che contiene informazioni sullo stato del processo, il suo program counter, lo stack pointer, l'allocazione di memoria, lo stato dei suoi file aperti, le informazioni necessarie per l'addebito dei costi e per la schedulazione, e qualunque altra cosa debba essere salvata quando il processo passa da uno stato di *esecuzione* ad uno stato di *pronto* o *bloccato*, in maniera che possa essere fatto ripartire più tardi come se non fosse mai stato fermato.

La Figura 2.4 mostra alcuni dei campi più importanti di un blocco di controllo in un sistema tipico: i primi campi sono relativi alla gestione dei processi, mentre le altre due colonne riguardano, rispettivamente, la gestione della memoria e dei file. Si noti che i campi precisi di una tabella di processi dipendono fortemente dal sistema, ma questa figura dà un'idea generale del tipo di informazione necessaria.

Ora che abbiamo visto la tabella dei processi, è possibile spiegare un po' meglio come è possibile mantenere l'illusione di una molteplicità di processi sequenziali in una macchina con una sola CPU e molti dispositivi di ingresso/uscita. Ad ogni classe di dispositivi ingresso/uscita (ad esempio dischetti, dischi rigidi, timer, terminali) viene associata una locazione, spesso vicina alla parte bassa della memoria, chiamata **vettore delle interruzioni (interrupt vector)**, che contiene l'indirizzo della procedura di gestione delle interruzioni. Supponiamo che sia in esecuzione il processo utente 3, quando si verifica un'interruzione dal disco: il program counter di questo processo, la parola di stato del programma e magari uno o più registri vengono messi sullo stack corrente dall'**hardware dedicato alle interruzioni**, e la CPU salta all'indirizzo specificato nel vettore delle interruzioni. Questo è tutto quello che fa l'**hardware**; da qui in poi, è tutto in mano al software, in particolare della procedura di servizio dell'interrupt.

Tutte le interruzioni cominciano con il salvataggio dei registri, spesso nell'elemento della tabella dei processi relativo al processo corrente; successivamente, l'informazione depositata sullo stack dall'interruzione viene rimossa e lo stack pointer viene fatto puntare ad uno stack temporaneo usato dal gestore dei processi. Azioni come il salvataggio dei registri o l'inizializzazione dello stack pointer non possono essere espresse in linguaggi ad alto livello come il C, quindi vengono svolte da una piccola routine in linguaggio assembler, normalmente la stessa per tutte le interruzioni, dal momento che il lavoro di salvare i registri è identico, non importa quale sia la causa dell'interruzione.

Quando questa routine termina, chiama una procedura C per eseguire il resto del lavoro per lo specifico tipo di interruzione (supponiamo che il sistema operativo sia scritto in C, la scelta generalmente fatta da tutti i sistemi operativi reali). Quando questa ha svolto il suo compito, per esempio facendo in modo che qualche processo passi allo stato di pronto, viene richiamato lo scheduler per vedere quale è il prossimo processo da eseguire. Dopodiché, il controllo viene passato al codice in linguaggio assembler perché vengano caricati i registri e la mappa di memoria per il nuovo processo corrente, che vie-

Gestione dei processi	Gestione della memoria	Gestione dei file
Registri	Puntatore al segmento di testo	Directory radice
Program counter	Puntatore al segmento dati	Directory di lavoro
Parola di stato del programma	Puntatore al segmento dello stack	Descrittori dei file
Stack pointer		Identificatore dell'utente (UID)
Stato del processo		Identificatore del gruppo (GID)
Priorità		
Parametri di schedulazione		
Identificatore del processo		
Processo genitore		
Gruppo del processo		
Segnali		
Tempo al quale è stato fatto partire il processo		
Tempo di CPU usato		
Tempo di CPU dei figli		
Tempo del prossimo allarme		

Figura 2.4 Alcuni dei campi di un elemento tipico della tabella dei processi.

1. L'hardware mette sullo stack il program counter ecc.
2. L'hardware carica un nuovo program counter dal vettore delle interruzioni
3. Una procedura in linguaggio assembler salva i registri
4. Una procedura in linguaggio assembler prepara il nuovo stack
5. Viene eseguita una procedura di servizio per le interruzioni scritta in C (solitamente, legge i dati in ingresso e li mette in un buffer)
6. Lo scheduler decide qual è il processo successivo da mandare in esecuzione.
7. La procedura in C restituisce il controllo al codice assembler.
8. La routine in codice assembler manda in esecuzione il processo corrente.

Figura 2.5 Schema delle azioni del livello più basso del sistema operativo quando si verifica un'interruzione.

ne fatto ripartire. La gestione dell'interruzione e la schedulazione sono riassunte in Figura 2.5. Si noti che i dettagli possono variare da sistema a sistema.

2.2 Thread

Nei sistemi operativi tradizionali, ogni processo ha uno spazio di indirizzamento e un singolo thread (letteralmente, filo) di controllo: in effetti, questa è quasi la definizione di processo. Ciononostante, ci sono spesso situazioni in cui è desiderabile avere molti thread di controllo nello stesso spazio di indirizzamento, in esecuzione quasi in parallelo, come se fossero processi separati (eccetto per lo spazio di indirizzamento condiviso). Nelle sezioni seguenti discuteremo queste situazioni e le loro implicazioni.

2.2.1 Il modello a thread

Il modello a processi che abbiamo discusso finora è basato su due concetti indipendenti: raggruppamento delle risorse e esecuzione. Qualche volta è utile separarli ed è qui che entrano in scena i thread.

Un processo può essere visto come un modo per raggruppare risorse tra loro correlate. Un processo ha uno spazio di indirizzamento che contiene il testo del programma e i dati, ed altre risorse, come file aperti, processi figli, alarmi incombenti, gestori di segnali, informazioni di contabilità ed altro ancora; raggruppate sotto forma di processo, queste risorse possono essere gestite più facilmente.

L'altro concetto che un processo racchiude è quello di thread di esecuzione, normalmente abbreviato con thread. Il thread ha un program counter che tiene traccia della prossima istruzione da eseguire, ha dei registri, che mantengono le variabili di lavoro correnti. Ha uno stack, che contiene la storia dell'esecuzione, con un elemento per ogni procedura che è stata chiamata ma non è ancora terminata. Sebbene un thread debba essere in esecuzione in qualche processo, il thread e il suo processo sono concetti diversi e possono essere trattati separatamente. I processi vengono usati per raggruppare le risorse, mentre i thread sono le entità schedulate per l'esecuzione nella CPU.

Quello che i thread aggiungono al modello a processi è il permettere molte esecuzioni nell'ambiente di un processo, in larga misura indipendenti l'una dall'altra. Avere più thread che vengono eseguiti in parallelo in un solo processo è analogo ad avere vari processi che sono in esecuzione in parallelo su un singolo calcolatore. Nel primo caso, i thread condividono uno spazio di indirizzamento, i file aperti e altre risorse, mentre nel secondo caso, i processi condividono la memoria fisica, i dischi, le stampanti e altre risor-

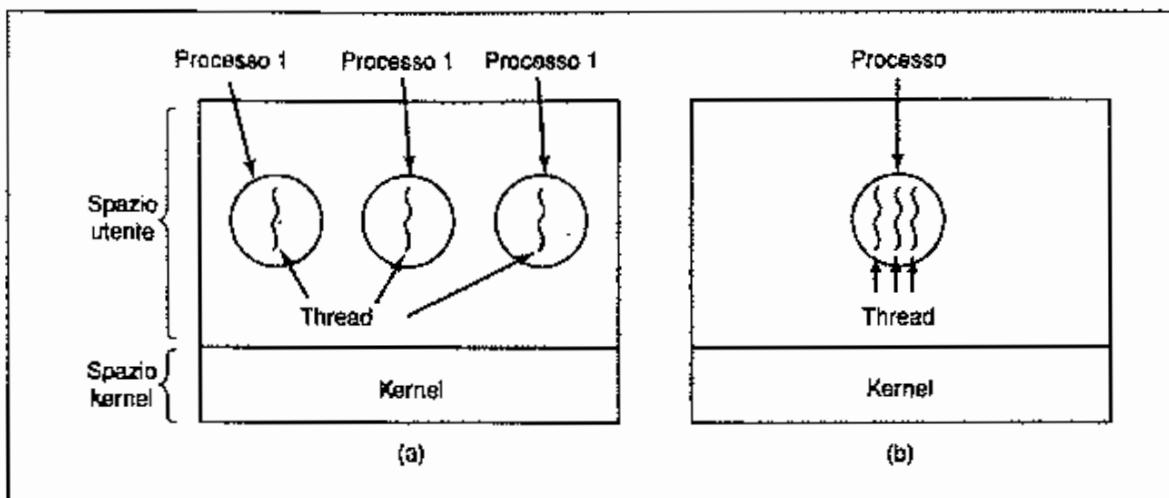


Figura 2.6 [a] Tre processi, ciascuno con un thread. [b] Un processo con tre thread.

se. Poiché i thread hanno solo alcune delle proprietà dei processi, a volte vengono chiamati **processi leggeri**. Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un solo processo sono associati più thread.

In Figura 2.6(a) vediamo tre processi tradizionali. Ogni processo ha il proprio spazio di indirizzamento e un solo thread di controllo, mentre in Figura 2.6(b) vediamo un solo processo con tre thread di controllo. Sebbene in entrambi i casi abbiamo tre thread, in Figura 2.6(a) ognuno di essi opera in spazi di indirizzamento diversi, mentre in Figura 2.6(b) tutti e tre condividono lo stesso spazio di indirizzamento.

Quando un processo con thread multipli viene eseguito su un sistema con una sola CPU, i thread vengono eseguiti a turno. In Figura 2.1 abbiamo visto come lavora la multiprogrammazione dei processi: effettuando il cambio di contesto fra vari processi, il sistema dà l'illusione di processi separati e sequenziali che vengono eseguiti in parallelo; il multithreading lavora nello stesso modo: la CPU passa rapidamente avanti e indietro tra i thread, dando l'illusione che i thread siano in esecuzione in parallelo, sebbene su una CPU più lenta di quanto non lo sia in realtà. Tre thread che eseguono elaborazioni intensive, sembrano eseguiti in parallelo, ciascuno su una CPU con un terzo della velocità della CPU reale.

Thread diversi in un processo non sono indipendenti come processi diversi, perché tutti i thread hanno esattamente lo stesso spazio di indirizzamento, cioè condividono anche le stesse variabili globali. Dal momento che ogni thread può accedere ad ogni indirizzo di memoria nello spazio di indirizzamento del processo, un thread può leggere, scrivere o per-

Voci relative a ciascun processo	Voci relative ad ogni thread
Spazio di indirizzamento	Program counter
Variabili globali	Registri
File aperti	Stack
Processi figli	Stato
Alarmi incombenti	
Segnali e gestori di segnali	
Informazioni di contabilità	

Figura 2.7 Nella prima colonna sono elencati alcuni elementi condivisi da tutti i thread di un processo. Nella seconda sono elencati alcuni elementi privati di ciascun thread.

sino cancellare completamente lo stack di un altro thread: non c'è protezione tra i thread, perché (1) è impossibile realizzarla e (2) non dovrebbe essere necessaria. Diversamente da processi diversi, che possono provenire da diversi utenti e potrebbero essere dannosi l'un l'altro, un processo appartiene sempre a un solo utente, che, presumibilmente, ha creato thread multipli in modo che possano cooperare, non combattere. Oltre alla condivisione dello spazio di indirizzamento, tutti i thread condividono lo stesso insieme di file aperti, processi figli, allarmi, segnali, ecceterà, come mostrato in Figura 2.7. Così, l'organizzazione di Figura 2.6(a) potrebbe essere usata quando tre processi sono fondamentalmente scorrelati, mentre la Figura 2.6(b) potrebbe essere appropriata quando i tre thread sono veramente parte dello stesso job e cooperano l'un l'altro attivamente e da vicino.

Le voci nella prima colonna sono proprietà dei processi, non dei thread: ad esempio, se un thread apre un file, quel file è visibile agli altri thread che appartengono al processo, che possono leggerlo e scriverlo. Questo è logico, dal momento che il processo è l'unità di gestione delle risorse e non il thread, mentre se ogni thread avesse il proprio spazio degli indirizzi, file aperti, allarmi incombenti e così via, sarebbe un processo separato. Quello che stiamo cercando di conseguire con il concetto di thread è la capacità dei thread di esecuzione multipli di condividere un insieme di risorse, così che possano lavorare bene insieme per eseguire una qualche mansione.

Come un processo tradizionale (cioè, un processo con un solo thread), un thread può essere in uno qualsiasi di diversi stati: in esecuzione, bloccato, pronto o terminato. Un thread in esecuzione ha il possesso della CPU ed è attivo; un thread bloccato sta aspettando qualche evento che lo sblocchi, ad esempio, quando un thread esegue una chiamata di sistema per leggere dalla tastiera, è bloccato finché l'input non è stato battuto. Un thread si può bloccare, aspettando il verificarsi di qualche evento o che qualche altro thread lo sblocchi; mentre un thread in stato di pronto è programmato per essere eseguito, e lo sarà presto, non appena arriverà il suo turno. Le transizioni fra gli stati dei thread sono le stesse che esistono fra gli stati dei processi e sono illustrate in Figura 2.2.

È importante capire che ogni thread ha il proprio stack, come mostrato in Figura 2.8. Lo stack di ciascun thread contiene un elemento per ogni procedura chiamata ma non ancora conclusa, con le variabili locali alla procedura e gli indirizzi di ritorno da usare quando la chiamata di procedura è terminata. Ad esempio, se la procedura X chiama la procedura Y e questa richiama la procedura Z, mentre Z è in esecuzione gli elementi per X, Y e Z saranno tutti sullo stack. Ogni thread chiamerà generalmente procedure diverse e avrà quindi una diversa storia di esecuzione: questo è il motivo per cui ogni thread necessita del proprio stack.

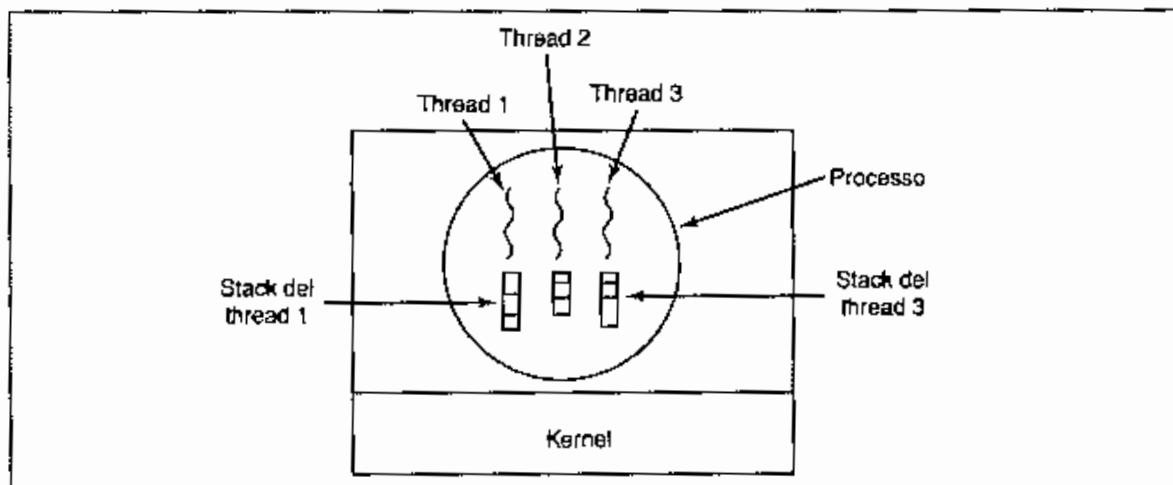


Figura 2.8 Ogni thread ha il proprio stack.

Quando è presente il multithreading, normalmente i processi partono con un solo thread presente, che ha la capacità di creare nuovi thread richiamando una procedura di libreria, come, ad esempio, *thread_create* e un parametro che tipicamente specifica il nome della procedura che il nuovo thread deve eseguire. Non è necessario (né possibile) specificare qualcosa sul nuovo spazio di indirizzamento del thread, dal momento che questo, automaticamente, è in esecuzione nello spazio di indirizzamento del thread che l'ha creato. Qualche volta i thread sono gerarchici, con una relazione genitore-figlio, ma spesso non esiste nessuna relazione e tutti i thread sono equivalenti. Con o senza relazione gerarchica, al primo thread viene normalmente restituito l'identificatore del thread creato, che permette di riferirsi per nome al nuovo thread.

Quando un thread ha finito il suo lavoro, può terminare richiamando una procedura di libreria, diciamo *thread_exit*, dopodiché scompare e non è più schedulabile per l'uso della CPU. In alcuni sistemi di thread, un thread può aspettare che uno (specifico) thread termini, richiamando una procedura, per esempio *thread_wait*. Questa procedura blocca il thread che l'ha richiamata fino a che uno (specifico) thread non è terminato. A questo proposito, la creazione e la terminazione dei thread è molto simile alla creazione e alla terminazione dei processi, con quasi le stesse opzioni.

Un'altra chiamata comune per i thread è *thread_yield*, che permette a un thread di cedere la CPU a un altro thread, perché venga eseguito. Tale chiamata è importante perché non esiste un'interruzione di clock per realizzare veramente il timesharing, come esiste invece per i processi, così è importante che i thread siano educati e cedano di quando in quando volontariamente la CPU, dando agli altri thread la possibilità di essere eseguiti. Altre chiamate permettono a un thread di aspettare che un altro thread finisca qualche lavoro, oppure di annunciare che ha finito qualche lavoro, e così via.

Mentre i thread spesso si rivelano utili, introducono un certo numero di complicazioni nel modello di programmazione. Per cominciare, consideriamo l'effetto della chiamata di sistema *fork* di UNIX. Se il processo genitore ha thread multipli, anche il figlio dovrebbe averne? Se così non è, il processo potrebbe non funzionare correttamente, dal momento che sono tutti essenziali.

Comunque, se il processo figlio ottiene tanti thread quanti il genitore, cosa accade se un thread nel genitore viene bloccato su una chiamata di *read*, diciamo dalla tastiera? Sono bloccati due thread, uno nel processo genitore e uno nel processo figlio? Quando viene battuta una riga, i due thread devono averne una copia? Solo il genitore? Solo il figlio? Lo stesso problema si ha con le connessioni di rete aperte.

Un'altra classe di problemi è collegata al fatto che i thread condividono molte strutture dati. Cosa accade se un thread chiude un file mentre un altro thread sta ancora leggendolo? Supponete che un thread rilevi che c'è troppa poca memoria e cominci ad allocarne dell'altra, e che a metà si verifichi un cambio di thread e anche il nuovo thread noti che c'è troppa poca memoria e cominci ad allocarne dell'altra: la memoria verrebbe probabilmente allocata due volte. Questi problemi possono essere risolti, con un po' di sforzo, ma occorrono una riflessione e un progetto attenti per fare in modo che i programmi multithread lavorino correttamente.

2.2.2 Uso dei thread

Avendo descritto cosa sono i thread, è tempo di spiegare perché tutti li vogliono. Il motivo principale per avere i thread è che in molte applicazioni, attività diverse procedono contemporaneamente e alcune di queste possono bloccarsi di tanto in tanto. Decomponendo un'applicazione di questo tipo in diversi thread sequenziali eseguiti quasi in parallelo, il modello di programmazione si semplifica.

Abbiamo già visto questa motivazione in precedenza: è la stessa che rende necessari i processi. Invece di pensare ad interruzioni, temporizzatori e cambi di contesto, possiamo pensare a processi paralleli. Solo che, adesso, con i thread, viene aggiunto un nuovo elemento: la capacità di entità parallele di condividere tra loro uno spazio di indirizzamento e tutti i suoi dati. Questa capacità è indispensabile per certe applicazioni, nelle quali l'utilizzo di processi multipli (con i loro spazi di indirizzamento separati) non funzionerebbe.

Una seconda motivazione per avere i thread è che possono essere creati e distrutti più facilmente rispetto ai processi, poiché non hanno alcuna risorsa associata: in molti sistemi la creazione di un thread è 100 volte più veloce della creazione di un processo. Questa proprietà è molto utile quando si ha bisogno che il numero di thread cambi dinamicamente e rapidamente.

Una terza ragione per avere i thread sono le prestazioni: i thread non portano alcun guadagno di prestazioni quando sono tutti dediti a calcoli intensivi (CPU bound), ma quando ci sono calcoli considerevoli e anche numerose operazioni di ingresso/uscita, l'utilizzo di thread permette la sovrapposizione di queste attività, velocizzando l'applicazione.

Infine, i thread sono utili su sistemi con CPU multiple, dove è possibile il parallelismo reale. Torneremo su questo argomento nel Capitolo 8.

È probabilmente più facile capire perché i thread sono utili utilizzando qualche esempio concreto. Come primo esempio consideriamo un elaboratore di testo: la maggior parte di essi visualizza sullo schermo il documento che sta creando, formattato esattamente come apparirà sulla pagina stampata. In particolare, tutte le interruzioni di linea e di pagina sono nella loro corretta posizione finale, così che l'utente li possa esaminare e cambiare, se necessario (per esempio per eliminare vedove ed orfanù, cioè linee incomplete al principio o alla fine della pagina, che sono considerate esteticamente sgradevoli).

Supponiamo che l'utente stia scrivendo un libro: dal punto di vista dell'autore, è facile tenere l'intero libro in un unico file per semplificare la ricerca di argomenti, eseguire sostituzioni globali e così via. In alternativa, ogni capitolo può essere in un file separato. Tuttavia, avere ogni sezione e sottosezione in file separati è una vera seccatura, quando devono essere fatti cambiamenti globali sull'intero libro, poiché bisogna modificare manualmente centinaia di file. Per esempio, se lo standard proposto xxxx è stato approvato appena prima che il libro vada in stampa, tutte le occorrenze di "Proposta di Standard xxxx" devono essere cambiate in "Standard xxxx" all'ultimo minuto. Se l'intero libro è in un file, di solito basta un singolo comando per eseguire tutte le sostituzioni, ma se il libro è disperso in più di 300 file, ognuno di essi deve essere modificato separatamente.

Consideriamo ora cosa succede quando l'utente cancella improvvisamente una frase dalla prima pagina di un documento di 800 pagine. Dopo aver controllato che la pagina modificata sia corretta, l'utente vuole ora fare un'altra modifica a pagina 600 e digita un comando che chiede all'elaboratore di testo di spostarsi su quella pagina (ad esempio, cercando una frase che compare solo lì). L'elaboratore di testo è ora forzato a riformattare sui due piedi l'intero libro fino a pagina 600, perché non sa quale sarà la prima linea di tale pagina fino a che non ha elaborato tutte le pagine precedenti. Si può verificare un ritardo consistente prima che la pagina 600 possa essere mostrata, portando ad avere un utente scontento.

In questo caso i thread possono aiutare. Supponiamo che l'elaboratore di testo sia scritto come un programma con due thread: un thread interagisce con l'utente e l'altro gestisce la formattazione in background. Non appena la frase è stata cancellata dalla prima pagina, il thread interattivo chiede al thread di formattazione di riformattare l'intero libro. Nel frattempo, il thread interattivo continua ad ascoltare il mouse e la tastiera e a rispondere a semplici comandi come scorrere la prima pagina, mentre l'altro thread lavora come un pazzo in background. Con un po' di fortuna, la riformattazione sarà completata prima che l'utente chieda di vedere la pagina 600, che potrà essere mostrata istantaneamente.

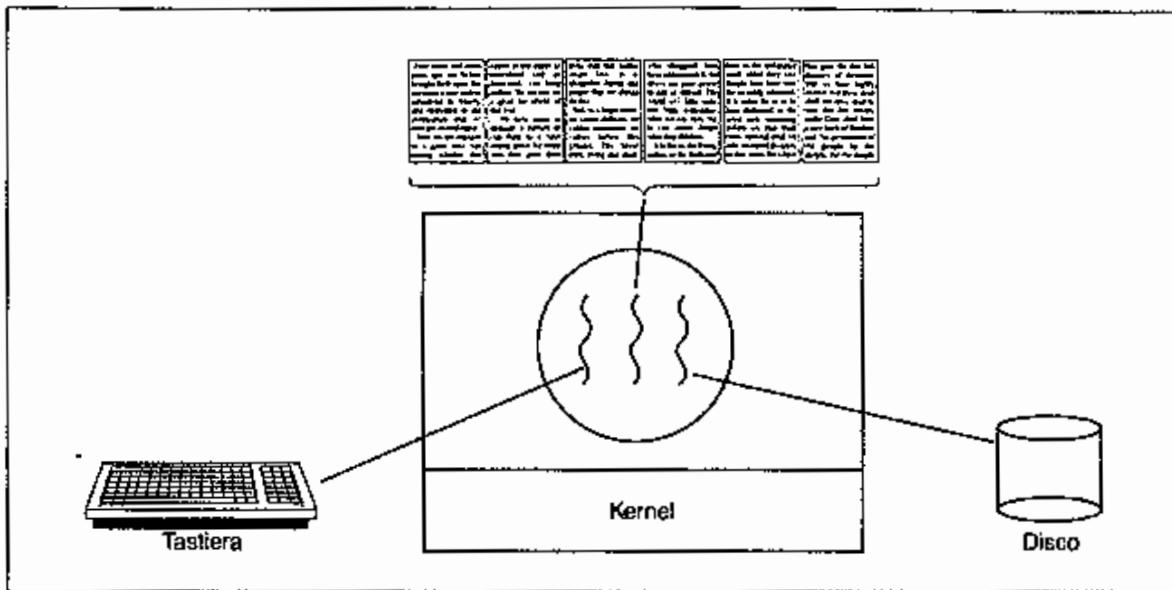


Figura 2.9 Un elaboratore di testo con tre thread.

Già che ci siamo, perché non aggiungere un terzo thread? Molti elaboratori di testo possono automaticamente salvare tutto il file su disco ogni pochi minuti per proteggere l'utente dalla perdita del lavoro del giorno in caso di crash del programma o del sistema, oppure di interruzione dell'energia elettrica. Il terzo thread può gestire i salvataggi di sicurezza su disco senza interferire con gli altri due. La situazione con i tre thread è mostrata in Figura 2.9.

Se il programma fosse composto da un solo thread, ogni volta che parte un salvataggio di sicurezza i comandi dalla tastiera e dal mouse verrebbero ignorati fino a quando l'operazione non è terminata, e l'utente percepirebbe il tutto con una prestazione lenta. In alternativa, gli eventi prodotti dalla tastiera e dal mouse potrebbero interrompere il salvataggio su disco, permettendo buone prestazioni, ma portando a un modello di programmazione complesso, guidato dalle interruzioni. Con tre thread, il modello di programmazione è molto più semplice: il primo thread interagisce solo con l'utente, il secondo thread riformatta il documento quando gli viene richiesto e il terzo thread scrive periodicamente il contenuto della RAM su disco.

Dovrebbe essere chiaro che avere tre processi separati in questo caso non funzionerebbe, perché tutti e tre i thread hanno bisogno di operare sul documento, mentre tre thread possono condividere la stessa memoria e quindi avere tutti accesso al documento.

Un'altro esempio analogo si ha con molti altri programmi interattivi. Per esempio, un foglio elettronico è un programma che permette all'utente di gestire una matrice, in cui alcuni elementi sono dati forniti dall'utente stesso. Altri elementi sono calcolati in base ai dati in ingresso usando funzioni potenzialmente complesse. Quando un utente modifica un elemento, potrebbe essere necessario ricalcolarne molti altri. Utilizzando un thread in background che riesegue i calcoli, il thread interattivo permetterebbe all'utente di effettuare ulteriori modifiche mentre il calcolo procede. In modo simile, un terzo thread può gestire da solo i salvataggi di sicurezza periodici.

Ora consideriamo ancora un altro esempio in cui i thread sono utili: un server per un sito World Wide Web, al quale arrivano richieste di pagine che vengono spedite al client. Nella maggior parte dei siti Web, alcune pagine sono accedute più di frequente di altre. Per esempio, la home page del sito della Sony è acceduta molto più spesso che non una pagina interna contenente le specifiche tecniche di una particolare videocamera. I server

Web sfruttano questo fatto per aumentare le prestazioni, mantenendo in memoria principale un collezione di pagine usate frequentemente, per eliminare la necessità di accedere al disco per recuperarle. Questa collezione è chiamata **cache** ed è usata anche in molti altri contesti.

Un modo di organizzare un server Web è mostrato in Figura 2.10. Qui un thread, il **dispatcher**, legge dalla rete le richieste da elaborare in arrivo; dopo averle esaminate, sceglie un **thread lavoratore** inattivo (per esempio bloccato) e gli consegna la richiesta, ad esempio scrivendo un puntatore al messaggio in una parola speciale associata ad ogni thread, dopodiché il dispatcher sveglia il lavoratore inattivo, spostandolo dallo stato bloccato allo stato pronto.

Quando il lavoratore si sveglia, controlla se la richiesta può essere soddisfatta da una pagina Web in cache, alla quale hanno accesso tutti i thread; se non è così, inizia un'operazione di read per ottenere la pagina dal disco e si blocca fino a che l'operazione non termina. Quando il thread si blocca sull'operazione di lettura da disco, viene scelto un altro thread da eseguire, per esempio il dispatcher per acquisire altro lavoro, oppure un altro lavoratore che è pronto ad essere eseguito.

Questo modello permette di scrivere il server come una collezione di thread sequenziali: il programma dispatcher è costituito da un ciclo infinito che ottiene una richiesta e la passa a un lavoratore, mentre il codice di ogni lavoratore consiste in un ciclo infinito che accetta una richiesta dal dispatcher e controlla se nella cache è presente la pagina richiesta e, se così, la pagina viene restituita al client e il lavoratore si blocca in attesa di una nuova richiesta, altrimenti legge la pagina dal disco, la restituisce al client e si blocca in attesa di una nuova richiesta.

Un abbozzo approssimativo del codice è dato in Figura 2.11. Qui, come nel resto del libro, si suppone che **TRUE** sia la costante 1. Inoltre **buf** e **pagina** sono strutture appropriate per memorizzare rispettivamente una richiesta e una pagina Web.

Consideriamo come il server Web possa essere scritto in assenza di thread: una possibilità è che operi come un singolo thread, il ciclo principale del server web legge una richiesta, la esamina e la completa prima di poter accettare la prossima, e mentre è in attesa della lettura da disco, il server è inattivo e non può elaborare nessun'altra richiesta in arrivo. Se il server Web è in esecuzione su una macchina dedicata, come è normale, la CPU è semplicemente inutilizzata, mentre il server Web è in attesa della lettura da disco;

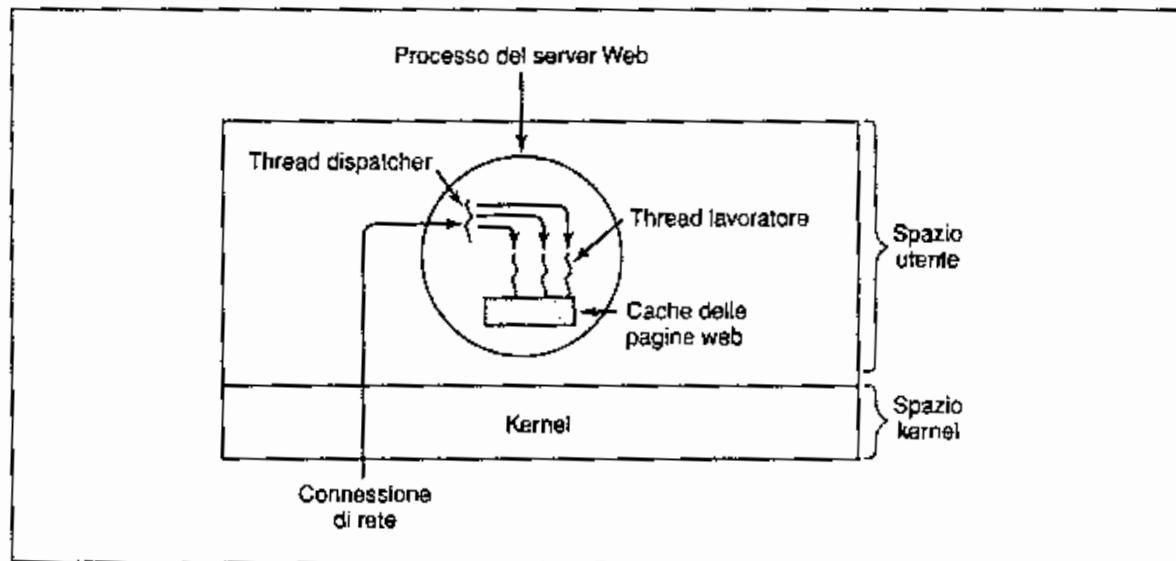


Figura 2.10 Un server Web a thread multipli.

```

while (TRUE) {
    accetta_richiesta(&buf);
    finisci_lavoro(&buf);
}

while (TRUE) {
    aspetta_lavoro(&buf)
    cerca_pagina_in_cache(&buf, &pagina);
    if (pagina_non_in_cache(&pagina))
        leggi_pagina_da_disco(&buf, &pagina);
    restituisce_pagina(&pagina);
}

```

(a) (b)

Figura 2.11 Un abbozzo approssimativo del codice per la Figura 2.11. (a) Thread dispatcher. (b) Thread lavoratore.

il risultato finale è che possono essere elaborate molte meno richieste al secondo. I thread fanno quindi aumentare considerevolmente le prestazioni, ma ogni thread è programmato sequenzialmente, nel solito modo.

Fino a qui abbiamo visto due possibili modelli: un server Web a thread multipli e un server Web ad un solo thread. Supponiamo che i thread non siano disponibili, ma che i progettisti del sistema trovino che la perdita di prestazioni dovuta a un singolo thread sia inaccettabile. Se è disponibile una versione non bloccante della chiamata di sistema read, è possibile un terzo approccio: quando arriva una richiesta, il solo e unico thread la esamina e se può essere soddisfatta dalla cache, bene, altrimenti viene iniziata un'operazione non bloccante su disco.

Il server memorizza lo stato della richiesta corrente in una tabella, quindi continua e ottiene il prossimo evento, che può essere la richiesta di un nuovo lavoro oppure una risposta dal disco a proposito della precedente operazione. Se è un nuovo lavoro, viene iniziato, mentre se è una risposta dal disco, le informazioni relative vengono prese dalla tabella e la richiesta viene elaborata. Con l'ingresso/uscita del disco non bloccante, una risposta dovrà avere probabilmente la forma di un segnale o di un'interruzione.

In questo schema, si perde il modello a "processo sequenziale" che avevamo nei primi due casi, perché lo stato della computazione deve essere salvato esplicitamente e ripristinato nella tabella ogni volta che il server cambia dal lavorare su una richiesta ad un'altra: in effetti, si stanno simulando i thread e il loro stack tra mille difficoltà. Un modello come questo, nel quale ogni computazione ha uno stato salvato ed esiste un insieme di eventi che possono accadere per cambiare lo stato, è chiamato **macchina a stati finiti**, ed è un concetto largamente usato in tutta l'informatica.

Ora dovrebbe essere chiaro cosa hanno da offrire i thread: rendono possibile mantenere l'idea di processo sequenziale che esegue chiamate di sistema bloccanti (per esempio per ingresso/uscita da disco) ed avere comunque parallelismo. Le chiamate di sistema bloccanti rendono più facile la programmazione ed aumentano le prestazioni. Il server ad un solo thread conserva la semplicità delle chiamate di sistema bloccanti, ma perde le prestazioni. Il terzo approccio raggiunge alte prestazioni attraverso il parallelismo,

Modello	Caratteristiche
Thread	Parallelismo, chiamate di sistema che bloccano
Processo a singolo thread	Nessun parallelismo, chiamate di sistema che bloccano
Macchina a stati finiti	Parallelismo, nessuna chiamata di sistema che blocca, interruzioni

Figura 2.12 I tre modi per costruire un server.

ma usa chiamate non bloccanti ed interruzioni e ciò è difficile da programmare. Questi modelli sono riassunti in Figura 2.12.

Un terzo esempio in cui i thread sono utili è un'applicazione che deve elaborare grandi quantità di dati. L'approccio normale è quello di leggere in ingresso un blocco di dati, elaborarlo e quindi scriverlo in uscita. Qui il problema è che il processo si blocca mentre i dati entrano ed escono, se sono disponibili solo chiamate di sistema bloccanti. Avere la CPU che rimane spesso inutilizzata quando ci sono molte elaborazioni da fare è chiaramente dispendioso e dovrebbe essere evitato, se possibile.

I thread offrono una soluzione. Il processo può essere strutturato con un thread per l'ingresso, un thread per l'elaborazione e un thread per l'uscita. Il thread per l'ingresso legge i dati in un buffer d'ingresso; il thread per l'elaborazione prende i dati dal buffer d'ingresso, li elabora e mette i risultati in un buffer d'uscita; il thread per l'uscita scrive questi dati di nuovo sul disco. In questo modo l'ingresso, l'uscita e l'elaborazione possono essere svolti contemporaneamente. Certamente questo modello funziona solo se le chiamate di sistema bloccano solo il thread che effettua la chiamata e non l'intero processo.

2.2.3 Implementazione dei thread nello spazio utente

Ci sono due modi principali per implementare i thread: nello spazio utente e nel kernel. La scelta è oggetto di discussione, ed è possibile anche un'implementazione ibrida; descriveremo questi metodi con i loro vantaggi e svantaggi.

Il primo metodo è mettere i thread interamente nello spazio utente, così il kernel non sa nulla di loro, e, non essendo coinvolto, gestirà processi ordinari ad un solo thread. Il primo ovvio vantaggio è che un thread a livello utente può essere implementato su sistemi operativi che non supportano i thread; tutti i sistemi utilizzati in passato ricadono in questa categoria, oltre ad alcuni utilizzati ancora oggi.

Tutte queste implementazioni hanno la stessa struttura generale, che è illustrata in Figura 2.13(a). I thread sono eseguiti sopra ad un sistema a tempo di esecuzione, che è una collezione di procedure che gestiscono i thread. Abbiamo già visto quattro di esse: *thread_create*, *thread_exit*, *thread_wait* e *thread_yield*, ma di solito ce ne sono anche altre.

Quando i thread sono gestiti nello spazio utente, ogni processo ha bisogno della propria tabella dei thread per tenere traccia dei propri thread. Questa tabella è analoga alla tabella dei processi del kernel, con la differenza che memorizza solo le proprietà collegate ai thread, come il program counter di ogni thread, lo stack pointer, i registri eccetera. La tabella dei thread è gestita dal sistema a tempo di esecuzione; quando un thread è spostato allo stato pronto o allo stato bloccato, le informazioni necessarie al suo riavvio sono memorizzate nella tabella dei thread, esattamente nello stesso modo in cui il kernel memorizza le informazioni sui processi nella tabella dei processi.

Quando un thread fa qualcosa che potrebbe farlo diventare bloccato localmente, per esempio in attesa che un altro thread nello stesso processo completi qualche lavoro, chiama una procedura del sistema a tempo di esecuzione, la quale controlla se il thread debba essere messo nello stato bloccato. Se è così, memorizza i registri del thread (cioè i propri) nella tabella dei thread, cerca nella tabella un thread pronto da eseguire e ricarica i registri macchina con i valori salvati del nuovo thread. Non appena lo stack pointer e il program counter sono stati scambiati, il nuovo thread ritorna vivo automaticamente. Se la macchina ha un'istruzione per memorizzare tutti i registri ed un'altra per caricarli tutti assieme, l'intero scambio dei thread può essere compiuto in una manciata di istruzioni. Compiere gli scambi di thread in questo modo è almeno un ordine di grandezza più rapido che usare le trap del kernel ed è una forte motivazione in favore dei thread package a livello utente.

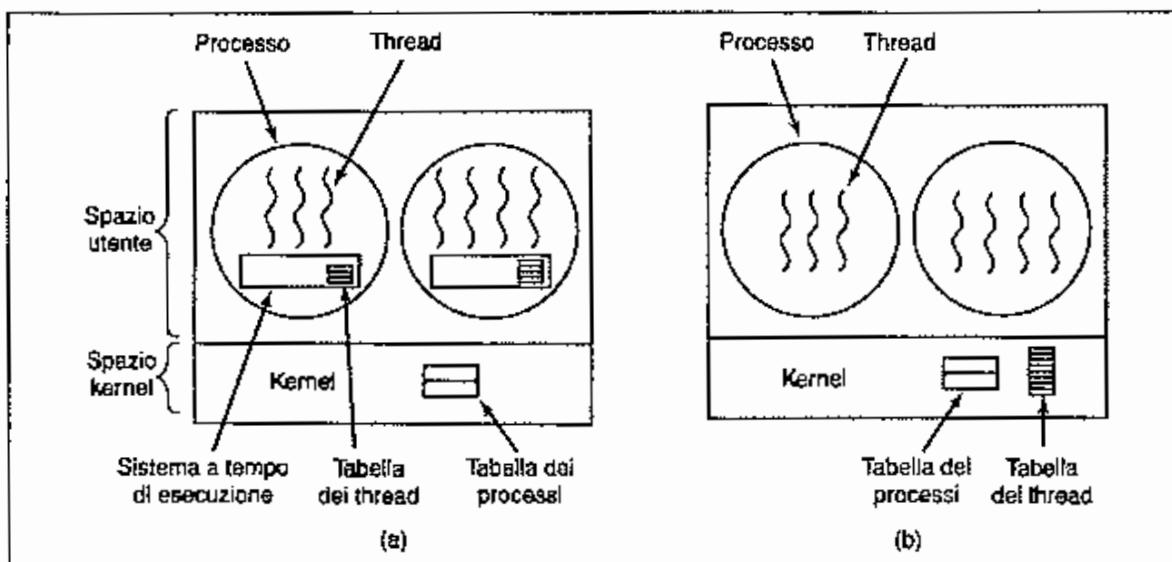


Figura 2.13 (a) Thread package a livello utente (b) Thread package gestito dal kernel.

Tuttavia, c'è una differenza chiave con i processi. Quando un thread ha finito la sua esecuzione per il momento, per esempio quando chiama *thread_yield*, il codice di *thread_yield* può esso stesso salvare le informazioni sul thread nella tabella; inoltre può richiamare lo scheduler dei thread facendogli scegliere un altro thread da eseguire. La procedura che salva lo stato del thread e lo scheduler sono procedure locali, quindi invocarle è molto più efficiente che fare una chiamata al kernel; inoltre, non è necessario usare le trap del kernel, né il cambio di contesto, la memoria cache non deve essere svuotata e così via. Tutto ciò rende la schedulazione dei thread molto veloce.

I thread a livello utente hanno anche altri vantaggi: permettono ad un processo di avere il proprio algoritmo di schedulazione. Per alcune applicazioni, per esempio quelle con un thread che implementa un *garbage collector*, è un vantaggio non doversi preoccupare che un thread venga fermato in un momento scomodo. Inoltre scalano meglio, poiché i thread a livello kernel richiedono invariabilmente spazio per le tabelle e per lo stack nel kernel, e ciò può essere un problema se il numero di thread è grande.

Nonostante le loro migliori prestazioni, i thread a livello utente hanno qualche grave problema. Il primo è come implementare le chiamate di sistema bloccanti: supponiamo che un thread legga dalla tastiera prima che un qualsiasi tasto sia stato premuto; lasciare che il thread esegua effettivamente la chiamata di sistema è inaccettabile, poiché bloccherà tutti i thread. Uno dei principali obiettivi dei thread era, in primo luogo, permettere ad ognuno di essi l'uso di chiamate bloccanti, ma impedire che un thread bloccato danneggiasse gli altri, e con le chiamate di sistema bloccanti, è difficile vedere come raggiungere questo obiettivo in modo semplice.

Le chiamate di sistema potrebbero essere tutte cambiate rendendole non bloccanti (per esempio una *read* sulla tastiera restituirebbe 0 byte se nessun carattere fosse già pronto nel buffer), ma richiedere modifiche al sistema operativo è antipatico. Per di più, una delle motivazioni per i thread a livello utente era proprio che essi potevano funzionare su qualunque sistema operativo esistente, e cambiare la semantica della *read* richiederebbe modifiche a molti programmi utente.

È possibile un'altra alternativa, nel caso in cui si possa dire in anticipo se una chiamata si bloccherà. In alcune versioni di UNIX, esiste una chiamata di sistema *select*, che permette di dire al chiamante se una eventuale *read* si bloccherà. Quando questa chiamata è presente, la procedura di libreria *read* può essere sostituita con una nuova, che pri-

ma esegue una chiamata alla `select` e solo dopo esegue la chiamata alla `read` se questa è sicura (cioè non si bloccherà). Se la `read` si bloccasse, la chiamata non verrebbe eseguita, ma verrebbe mandato in esecuzione un altro thread; la prossima volta che il sistema a tempo di esecuzione otterrà il controllo, potrà controllare ancora se la `read` è sicura. Questo approccio richiede la riscrittura di parte della libreria delle chiamate di sistema, è inefficiente e poco elegante, ma c'è poca scelta. Il codice messo attorno alla chiamata di sistema per eseguire i controlli è chiamato *jacket* (giacca) o *wrapper* (involturco).

Analogo al problema delle chiamate di sistema bloccanti è il problema dei fault di pagina, che verranno studiati nel Capitolo 4. Per il momento è sufficiente affermare che i calcolatori possono lavorare senza che tutto il programma sia presente nella memoria principale contemporaneamente. Se il programma chiama o salta ad un'istruzione che non è in memoria, si ha un fault di pagina e il sistema operativo andrà a prendere l'istruzione mancante (e quelle vicine) dal disco: questo viene detto fault di pagina. Mentre l'istruzione necessaria viene localizzata e letta, il processo è bloccato, e se un thread causa un fault di pagina, il kernel, non sapendo dell'esistenza del thread, naturalmente bloccherà l'intero processo fino a che l'ingresso/uscita del disco non verrà completata, anche se potrebbero andare in esecuzione altri thread.

Un altro problema con i thread a livello utente è che se un thread inizia l'esecuzione, nessun altro thread in quel processo verrà eseguito, a meno che il primo rilasci la CPU volontariamente. All'interno di un singolo processo non ci sono interruzioni di clock, rendendo impossibile la schedulazione sul modello round robin (con turni) dei processi. Se un thread non entra nel sistema a tempo di esecuzione di sua spontanea volontà, lo scheduler non ha alcuna possibilità.

Una possibile soluzione al problema dei thread che non terminano mai la propria esecuzione è avere un sistema a tempo di esecuzione che richieda un segnale di clock (interruzione) una volta al secondo per avere il controllo, ma questo è rozzo e complesso da programmare. Le interruzioni di clock periodiche a frequenze maggiori non sono sempre possibili e, anche quando lo sono, l'overhead totale può essere considerevole. Inoltre, anche un thread potrebbe aver bisogno di un'interruzione di clock, interferendo con il suo uso da parte del sistema a tempo di esecuzione.

Un'altra motivazione, e probabilmente la più scouvolgente contro i thread a livello utente, è che generalmente i programmatore vogliono i thread proprio nelle applicazioni dove i thread si bloccano spesso, come per esempio in un server Web a thread multipli. Questi thread fanno chiamate di sistema continuamente, ed una volta che avviene la trap al kernel per compiere la chiamata di sistema, il kernel non può far altro che cambiare thread, se il vecchio è bloccato, il che elimina il bisogno di richiamare costantemente la `select` per controllare se una chiamata di sistema `read` è sicura. Per quale motivo le applicazioni che fanno calcoli intensivi e si bloccano raramente dovrebbero utilizzare i thread? Nessuno proporrebbe seriamente di calcolare i primi n numeri primi o di giocare a scacchi utilizzando i thread, perché non c'è niente da guadagnarci facendolo così.

2.2.4 Implementazione dei thread nel kernel

Ora prendiamo in considerazione il caso in cui il kernel conosca e gestisca i thread: non serve un sistema a tempo di esecuzione, come mostrato in Figura 2.13(b), ed inoltre non c'è una tabella dei thread per ogni processo, ma il kernel ha una tabella dei thread che tiene traccia di tutti i thread nel sistema. Quando un thread desidera creare un nuovo thread o distruggere uno esistente, esegue una chiamata di sistema, che effettua la creazione o la distruzione aggiornando la tabella dei thread del kernel.

La tabella dei thread del kernel contiene i registri di ogni thread, lo stato e altre infor-

mazioni. Le informazioni sono le stesse dei thread a livello utente, ma ora sono nel kernel invece di essere nello spazio utente (all'interno del sistema a tempo di esecuzione). Queste informazioni sono un sottoinsieme di quelle tradizionalmente mantenute su ognuno dei processi a singolo thread, cioè lo stato del processo. Inoltre il kernel mantiene anche la tradizionale tabella dei processi per tenere traccia dei processi.

Tutte le chiamate che potrebbero bloccare un thread sono implementate come chiamate di sistema, con un costo considerevolmente più alto che una chiamata ad una procedura a tempo di esecuzione. Quando un thread si blocca, il kernel, a sua discrezione, può eseguire un altro thread appartenente allo stesso processo (se ce n'è uno pronto), oppure un thread di un altro processo. Con i thread a livello utente, il sistema a tempo di esecuzione mantiene in esecuzione i thread del proprio processo fino a che il kernel gli toglie l'uso della CPU (oppure non ci sono thread pronti).

A causa del costo relativamente alto della creazione e della distruzione di thread nel kernel, alcuni sistemi adottano un approccio corretto per l'ambiente, in modo da riciclare i loro thread. Quando un thread viene distrutto, è marcato come non eseguibile, ma le strutture corrispondenti nel kernel non sono modificate in nessun modo, così, quando dovrà essere creato un nuovo thread, il vecchio thread è riattivato, risparmiando un po' di overhead. Il riciclaggio è anche possibile per i thread a livello utente, ma poiché l'overhead per la loro gestione è molto più piccolo, c'è meno incentivo per farlo.

I thread nel kernel non richiedono nuove chiamate di sistema non bloccanti. Inoltre, se un thread in un processo causa un fault di pagina, il kernel può facilmente controllare se il processo ne ha altri eseguibili e, nel caso, eseguirne uno mentre aspetta che la pagina richiesta sia letta dal disco. Il loro principale svantaggio è che il costo di una chiamata di sistema è considerevole, perciò ci sarà molto più overhead se si hanno molte operazioni sui thread (creazione, terminazione, eccetera).

2.2.5 Implementazione ibrida

Sono stati studiati molti modi per cercare di combinare i vantaggi dei thread a livello utente con i thread a livello kernel. Un modo è utilizzare i thread a livello kernel e quindi mappare (multiplexing) molti thread a livello utente su alcuni o tutti i thread nel kernel, come mostrato in Figura 2.14.

In questo modello, il kernel si preoccupa solo dei thread a livello kernel e schedula

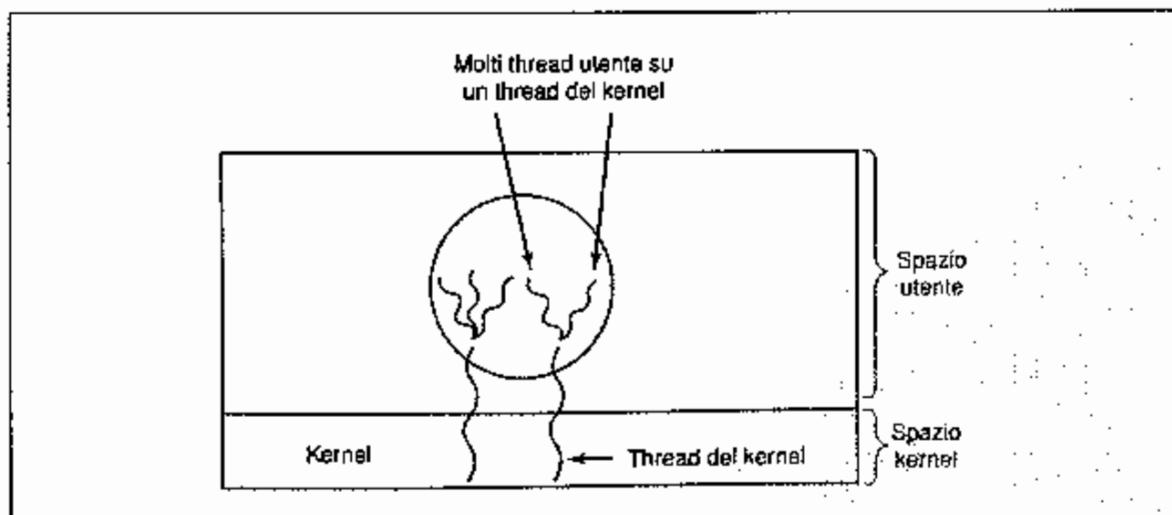


Figura 2.14 Multiplexing dei thread a livello utente su thread a livello kernel.

solo questi ultimi. Alcuni di essi possono avere più thread a livello utente mappati su di loro, che sono creati, distrutti e schedulati proprio come thread a livello utente in un processo che viene eseguito su un sistema senza la capacità di gestire thread multipli. In questo modello, ogni thread a livello kernel viene usato a turno da un certo insieme di thread a livello utente.

2.2.6 Attivazioni dello scheduler

Molti ricercatori hanno cercato di combinare i vantaggi dei thread a livello utente (buone prestazioni) con quelli dei thread nel kernel (non dover usare trucchetti per fare in modo che le cose funzionino). Di seguito verrà descritto uno di questi approcci eseguiti da Anderson e altri (1992), chiamato attivazione dello scheduler (scheduler activation). Un lavoro collegato è presentato da Edler et al. (1988) e Scott et al. (1990).

Gli obiettivi del lavoro chiamato attivazione dello scheduler è di imitare la funzionalità dei thread nel kernel, ma con prestazioni migliori e più flessibilità, di solito associate ai thread implementati nello spazio utente. In particolare i thread utente non dovrebbero utilizzare chiamate di sistema speciali non bloccanti, né controllare in anticipo se una certa chiamata di sistema è sicura. Nondimeno, quando un thread si blocca su una chiamata di sistema o un fault di pagina, dovrebbe essere possibile mandare in esecuzione un altro thread all'interno dello stesso processo, se ne esiste uno pronto.

L'efficienza è ottenuta evitando transizioni non necessarie tra spazio utente e spazio kernel. Se un thread si blocca in attesa che un altro thread faccia qualcosa, per esempio, non c'è ragione di coinvolgere il kernel, risparmiando l'overhead della transizione kernel-utente. Il sistema a tempo di esecuzione nello spazio utente può da solo bloccare il thread che si vuole sincronizzare, e schedularne uno nuovo.

Quando sono utilizzate le attivazioni dello scheduler, il kernel assegna un certo numero di processori virtuali ad ogni processo e lascia che sia il sistema a tempo di esecuzione (in spazio utente) ad allocare i processori per i vari thread. Questo meccanismo può anche essere utilizzato su un sistema a multiprocessore, dove i processori virtuali possono essere CPU reali. Il numero di processori virtuali assegnati ad un processo è inizialmente uno, ma il processo può richiederne ulteriori e rilasciare i processori di cui non ha più bisogno. Anche il kernel può riprendersi processori virtuali già allocati per assegnarli ad altri processi maggiormente bisognosi.

L'idea base che rende funzionante questo schema è che, quando il kernel sa che un thread è bloccato (per esempio ha eseguito una chiamata di sistema bloccante oppure ha causato un fault di pagina), lo notifica al processo del sistema a tempo di esecuzione, passandogli sullo stack il numero del thread in questione e una descrizione dell'evento accaduto. La notifica avviene con il kernel che attiva il sistema a tempo di esecuzione ad un indirizzo di partenza conosciuto, in modo approssimativamente analogo ai segnali in UNIX. Questo meccanismo viene chiamato upcall (letteralmente, chiamata verso l'alto).

Una volta attivato nel modo descritto, il sistema a tempo di esecuzione può rischedulare i propri thread, tipicamente marcando il thread corrente come bloccato e prendendone un altro dalla lista dei pronti, impostando i suoi registri e mandandolo in esecuzione. In seguito, quando il kernel saprà che il thread originale può essere eseguito nuovamente (per esempio la pipe dalla quale cercava di leggere, ora contiene dei dati, oppure la pagina che aveva causato il fault è stata letta dal disco), esegue un'altra upcall al sistema a tempo di esecuzione per informarlo dell'evento. Il sistema a tempo di esecuzione, a sua discrezione, può far ripartire il thread immediatamente, oppure metterlo nella lista dei pronti per essere eseguito in seguito.

Quando avviene un'interruzione hardware mentre viene eseguito un thread utente, la

CPU soggetta all'interruzione cambia nella modalità kernel. Se l'interruzione è causata da un evento che non è di interesse del processo interrotto, come il completamento dell'ingresso/uscita da parte di un altro processo, il gestore delle interruzioni, quando ha finito, rimette il thread interrotto nello stato in cui era prima dell'interruzione. Se, tuttavia, il processo è interessato all'interruzione, come all'arrivo della pagina necessaria ad uno dei suoi thread, il thread interrotto non viene fatto ripartire. Viene invece sospeso e viene fatto partire il sistema a tempo di esecuzione su quella CPU virtuale, con lo stato del thread interrotto sullo stack. È compito del sistema a tempo di esecuzione decidere quale thread schedulare su quella CPU: il thread interrotto, un nuovo thread pronto o qualche altra terza possibilità.

Un'obiezione all'attivazione dello scheduler è fondamentale: fidarsi delle upcall, un concetto che viola la struttura intrinseca di ogni sistema a strati. Normalmente lo strato n offre certi servizi che lo strato $n+1$ può richiamare, ma lo strato n non può chiamare procedure dello strato $n+1$; e le upcall non rispettano questo fondamentale principio.

2.2.7 Thread pop-up

I thread sono spesso utili nei sistemi distribuiti. Un esempio importante è la gestione dei messaggi in entrata, come le richieste di servizio: l'approccio tradizionale è avere un processo o un thread che si blocca sulla chiamata di sistema `receive`, mentre attende un messaggio in entrata, e quando arriva un messaggio, lo accetta e lo elabora.

Tuttavia, è possibile un approccio completamente diverso, nel quale l'arrivo di un messaggio causa la creazione da parte del sistema di un nuovo thread per gestirlo, chiamato **thread pop-up**, illustrato in Figura 2.15. Un vantaggio chiave dei thread pop-up è che non hanno alcuna storia, vale a dire registri, stack, eccetera da ripristinare, poiché sono nuovi di zecca, quindi ognuno inizia daccapo ed è identico a tutti gli altri, il che rende possibile crearli rapidamente. Al nuovo thread viene dato il messaggio in entrata da elaborare. Il risultato dell'uso dei thread pop-up è che la latenza tra l'arrivo di un messaggio e l'inizio dell'elaborazione può venire ridotta di molto.

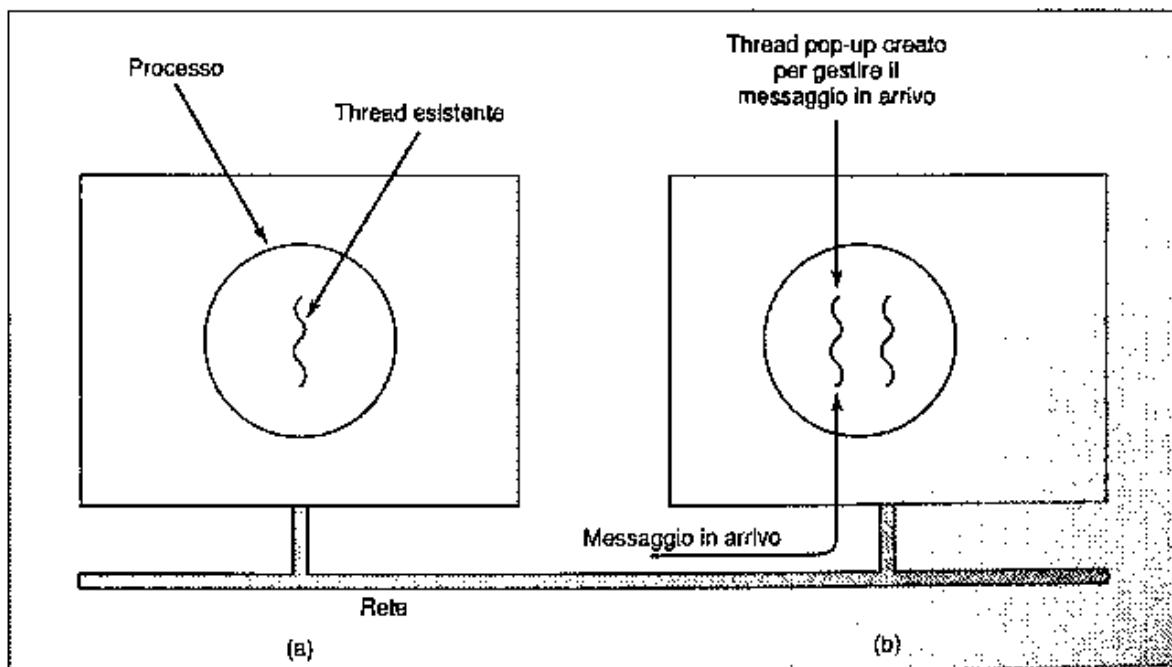


Figura 2.15 La creazione di un nuovo thread quando arriva un nuovo messaggio. [a] Prima che il messaggio arrivi. [b] Dopo che il messaggio è arrivato.

Quando si utilizzano i thread pop-up è necessaria una pianificazione preventiva. Per esempio, in quale processo verranno eseguiti i thread? Se il sistema supporta i thread nel contesto del kernel, i thread possono essere eseguiti qui (ecco perché nella Figura 2.15 non è stato mostrato il kernel). Di solito, avere i thread pop-up che vengono eseguiti nello spazio kernel è spesso più facile e veloce che metterli nello spazio utente, ed inoltre, un thread pop-up nello spazio kernel può accedere facilmente a tutte le tabelle del kernel stesso e ai dispositivi di ingresso/uscita, che potrebbero essere necessari per l'elaborazione delle interruzioni. D'altra parte, thread con bachi possono causare molti più danni se sono nel kernel, rispetto a quelli che possono causare se sono a livello utente. Per esempio, se un thread viene eseguito troppo a lungo e non c'è modo di togliergli il controllo della CPU, possono andare perduti dei dati in ingresso.

2.2.8 Trasformare codice a thread singolo in multithread

Molti programmi esistenti sono stati scritti per processi a thread singolo. La loro conversione ai thread multipli è molto più intricata di quello che potrebbe sembrare a prima vista. Di seguito verranno esaminati solo alcuni dei trabocchetti.

Come prima cosa, il codice di un thread è formato di solito da molte procedure, proprio come un processo. Può avere variabili locali, globali e parametri di procedura. Le variabili locali e i parametri non causano alcun problema, ma li causano le variabili che sono globali all'interno di un thread, ma non globali per l'intero programma. Queste sono variabili che sono globali, nel senso che molte procedure all'interno del thread le usano (come potrebbero usare qualsiasi variabile globale), ma gli altri thread dovrebbero logicamente lasciarle stare.

Come esempio, consideriamo la variabile *errno* mantenuta da UNIX. Quando un processo (o un thread) esegue una chiamata di sistema che fallisce, il codice d'errore è messo nella variabile *errno*. In Figura 2.16, il thread 1 esegue la chiamata di sistema *access* per verificare se ha il permesso di accedere a un certo file, ed il sistema operativo restituisce la risposta nella variabile globale *errno*. Dopo che il controllo è ritornato al thread 1, ma prima che abbia potuto leggere *errno*, lo scheduler decide che per il momento il thread 1 ha avuto abbastanza tempo di CPU e quindi di scambiarlo passando al thread 2, il quale esegue una chiamata a *open* che fallisce, causando la sovrascrittura di *errno* e la perdita per sempre del codice di accesso del thread 1. Quando, in seguito, il thread 1 ripartirà, leggerà un valore errato e si comporterà scorrettamente.

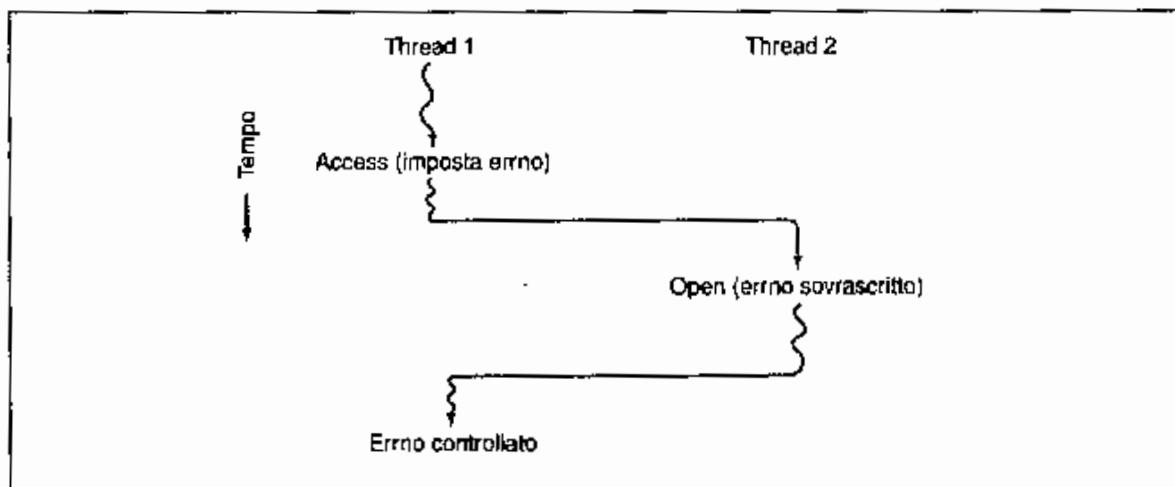


Figura 2.16 Conflitto fra thread sull'uso di una variabile globale.

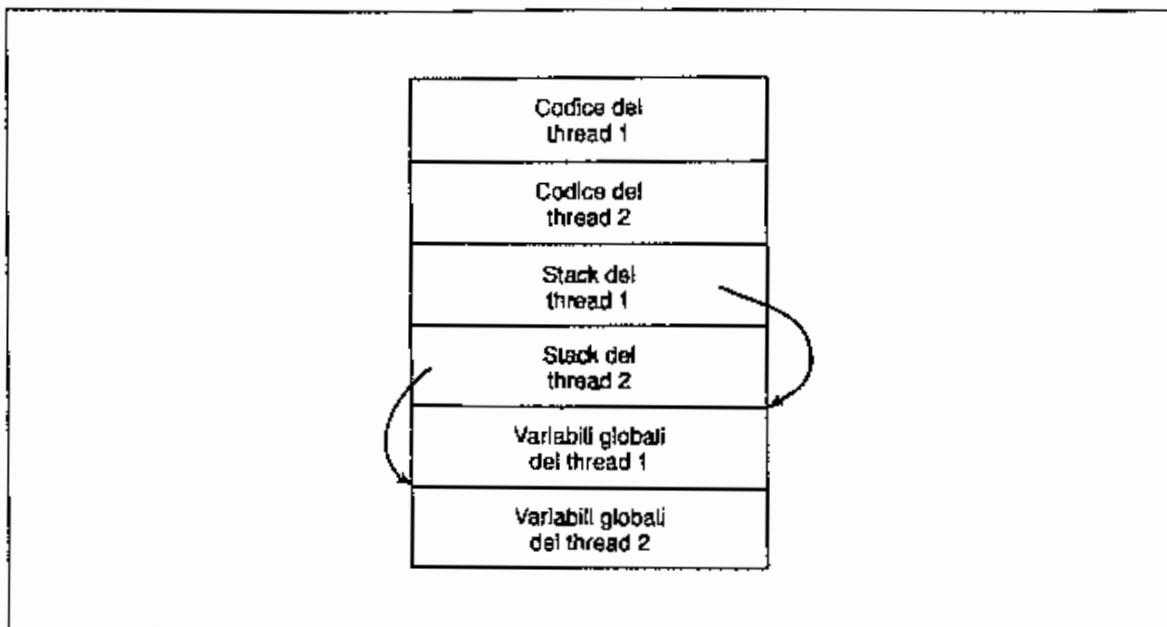


Figura 2.17 I thread possono avere variabili globali private.

Sono possibili varie soluzioni a questo problema. Una è proibire del tutto le variabili globali, ma, per quanto possa essere meritevole questo ideale, va in conflitto con molto software esistente. Un'altra è assegnare ad ogni thread le proprie variabili globali private, come mostrato in Figura 2.17: in questo modo, ogni thread ha la propria copia privata di *errno* e delle altre variabili globali, evitando i conflitti. In effetti, questa decisione crea un nuovo livello di scope (validità e visibilità delle variabili), cioè le variabili visibili a tutte le procedure di un thread, oltre ai livelli esistenti, variabili visibili solo a una procedura o ovunque nel programma.

L'accesso alle variabili globali private è un po' complesso, tuttavia, dato che la maggior parte dei linguaggi di programmazione ha un modo per esprimere variabili globali e locali, ma non le forme intermedie. È possibile allocare un pezzo di memoria per le variabili globali e passarla ad ogni procedura nel thread, come un parametro extra. Anche se è una soluzione poco elegante, funziona.

In alternativa, possono essere create nuove procedure di libreria, per creare, impostare e leggere queste variabili globali private dei thread. La prima chiamata potrebbe assomigliare a questa,

```
create_global("bufptr");
```

che alloca memoria per un puntatore chiamato *bufptr* sullo heap o in una speciale area di memorizzazione per il thread chiamante. Non importa dove la memoria è allocata, solo il thread chiamante ha accesso alla variabile globale. Se un altro thread crea una variabile globale con lo stesso nome, ottiene una locazione diversa che non va in conflitto con quella esistente.

Sono necessarie due chiamate per accedere alle variabili globali: una per scriverle e l'altra per leggerle. Per scrivere, qualcosa come questo,

```
set_global("bufptr", &buf);
```

che memorizza il valore di un puntatore nella locazione di memoria creata in precedenza dalla chiamata *create_global*. Per leggere una variabile globale, la chiamata potrebbe essere simile a questa,

```
bufptr = read_global("bufptr");
```

che restituisce l'indirizzo memorizzato nella variabile globale, così che si possa accedere ai suoi dati.

Un altro problema convertendo un programma a singolo thread in uno a thread multipli è che molte procedure di libreria non sono rientranti, cioè, non sono state progettate per gestire una seconda chiamata mentre la precedente non è ancora terminata. Per esempio, mandare un messaggio attraverso la rete potrebbe essere programmato assemblando il messaggio in un buffer fisso all'interno della libreria, quindi eseguendo una trap al kernel per spedirlo. Cosa succede se un thread assembla il suo messaggio nel buffer e poi un'interruzione di clock forza a passare ad un secondo thread, che sovrascrive immediatamente il buffer con il proprio messaggio?

In modo simile, le procedure che allocano memoria, come la *malloc* in UNIX, mantengono delle tabelle cruciali a proposito dell'uso della memoria, per esempio una lista concatenata di pezzi di memoria disponibili. Mentre la *malloc* è occupata nell'aggiornamento di queste liste, queste ultime possono essere in uno stato inconsistente, con puntatori che non puntano da nessuna parte. Se avviene un cambio di thread mentre le tabelle sono inconsistenti e arriva una nuova chiamata da parte di un altro thread, potrebbe essere usato un puntatore non valido, portando ad una caduta del programma. Sistemare tutti questi problemi in modo corretto significa, in pratica, riscrivere l'intera libreria.

Una diversa soluzione è fornire un jacket per ogni procedura di libreria, che imposta un bit per marcare che la libreria è in uso. Ogni tentativo di un altro thread di usare una procedura di libreria, mentre una precedente chiamata non è stata completata, verrà bloccato. Sebbene questo approccio possa funzionare, elimina moltissimo del parallelismo potenziale.

Inoltre, consideriamo i segnali: alcuni sono logicamente specifici di un thread, mentre altri non lo sono: per esempio, se un thread chiama *alarm*, ha senso che il segnale risultante vada al thread che ha eseguito la chiamata. Tuttavia, quando i thread sono implementati interamente nello spazio utente, il kernel, non essendo nemmeno a conoscenza dei thread, può difficilmente dirigere il segnale a quello giusto. Una ulteriore complicazione si ha se un processo può avere solo un allarme (o sveglia) sospeso per volta e più thread chiamano *alarm* indipendentemente.

Altri segnali, come interruzioni dalla tastiera, non sono specifici di un thread. Chi dovrebbe catturarli? Un thread designato? Tutti i thread? Un thread pop-up creato appositamente? Inoltre, cosa succede se un thread cambia il gestore dei segnali senza avvertire gli altri thread? E cosa succede se un thread vuole catturare un particolare segnale (per esempio la pressione di CTRL-C da parte dell'utente) e un altro vuole che tale segnale termini il processo? Questa situazione può sorgere se uno o più thread eseguono procedure di libreria standard ed altri sono scritti dall'utente; chiaramente questi desideri sono incompatibili. In generale, i segnali sono già abbastanza difficili da gestire in un ambiente a singolo thread, passare a un ambiente con thread multipli non li rende più facili da gestire.

Un ultimo problema introdotto dai thread è la gestione dello stack: in molti sistemi, quando un processo eccede la dimensione dello stack (stack overflow), il kernel semplicemente fornisce automaticamente più stack al processo. Quando un processo ha più thread, deve anche avere più stack, ma se il kernel non è consapevole di tutti questi stack, non può farli crescere automaticamente quando lo stack dà errore. In realtà, non

è nemmeno in grado di capire che un fault di memoria è collegato alla crescita dello stack.

Questi problemi non sono certo insormontabili, ma mostrano che introdurre semplicemente i thread in un sistema esistente, senza effettuare una consistente riprogettazione del sistema stesso, non funziona per niente: la semantica delle chiamate di sistema può essere ridefinita, e le librerie devono essere riscritte, a dir poco, e tutte queste cose devono essere fatte in modo tale da rimanere compatibili con i programmi esistenti, per il caso limite di un processo con un solo thread. Per ulteriori informazioni a proposito dei thread, si veda Hauser e altri, 1993 e Marsh e altri, 1991.

2.3 Comunicazione fra processi

Spesso i processi hanno bisogno di comunicare tra loro: ad esempio, in un pipeline di shell, i dati in uscita dal primo processo devono essere passati al secondo processo e così via lungo la linea. Quindi, c'è la necessità che i processi comunichino fra loro, preferibilmente in modo ben strutturato e senza utilizzare le interruzioni. Nelle sezioni che seguono esamineremo alcune delle questioni legate a questa **Comunicazione InterProcessi**, o **IPC** (**InterProcess Communication**).

Molto brevemente, ci sono tre questioni da considerare. La prima è stata anticipata in precedenza: come può un processo passare informazioni ad un altro? La seconda ha a che fare con l'essere sicuri che due o più processi non si mettano l'uno sulla strada dell'altro quando prendono parte ad attività critiche (supponete che due processi tentino entrambi di impadronirsi dell'ultimo 1MB di memoria). La terza riguarda il mettere in sequenza i processi in modo adeguato quando esistono delle dipendenze: se il processo A produce dei dati e il processo B li stampa, B deve aspettare finché A ha prodotto qualche dato prima di cominciare la stampa. Esamineremo tutti e tre questi problemi nella prossima sezione.

È anche importante accennare al fatto che due di queste questioni si applicano anche ai thread, tranne la prima, il passaggio di informazioni, che è facile per i thread dal momento che condividono uno spazio di indirizzamento comune, (thread in diversi spazi di indirizzamento che hanno bisogno di comunicare ricadono nella questione della comunicazione fra processi). Comunque, le altre due, lo stare fuori dagli spazi altrui e la scelta delle sequenze appropriate, si applicano in modo equivalente ai thread, che hanno gli stessi problemi, e per cui valgono le stesse soluzioni. Più avanti discuteremo il problema nel contesto dei processi, ma tenete a mente che gli stessi problemi e soluzioni si applicano anche ai thread.

2.3.1 Corse critiche

In alcuni sistemi operativi, i processi che lavorano insieme possono condividere una parte di memoria comune, che ciascuno può leggere e scrivere. Questo spazio condiviso può essere in memoria principale (magari in una struttura dati del kernel) oppure può essere un file condiviso; il tipo di memoria condivisa non cambia la natura della comunicazione o i problemi che ne conseguono. Per vedere come funziona praticamente la comunicazione tra processi, consideriamo un esempio semplice ma frequente: uno spooler di stampante. Quando un processo vuole stampare un file, aggiunge il nome del file in una directory speciale detta spooler directory o directory di spool; un altro processo, il demone della stampante, controlla periodicamente se ci sono file da stampare e, in tal caso, li stampa e rimuove i loro nomi dalla directory.

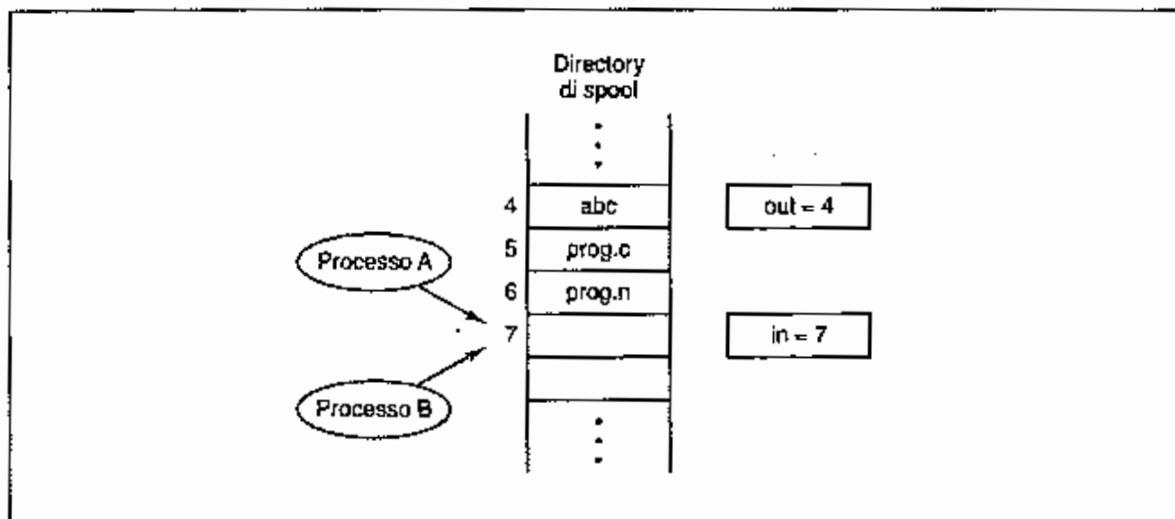


Figura 2.18 Due processi vogliono accedere alla memoria condivisa nello stesso istante.

Immaginiamo che la nostra directory di spool abbia un gran numero di elementi, numerati 0, 1, 2, ..., ciascuno capace di contenere il nome di un file. Immaginiamo anche che ci siano due variabili condivise, *out*, che punta al prossimo file da stampare, e *in*, che punta al prossimo elemento libero della directory; esse potrebbero essere tenute in un file di due parole, accessibile da tutti i processi. Ad un certo istante, gli elementi da 0 a 3 sono vuoti (i file sono già stati stampati) e gli elementi da 4 a 6 sono pieni (con i nomi dei file accodati per la stampa), quando, più o meno simultaneamente, i processi *A* e *B* decidono di accodare un file per la stampa. La situazione è illustrata nella Figura 2.18.

Nelle giurisdizioni dove è applicabile la legge di Murphy (*se qualcosa può andare storto, andrà storto*), potrebbe succedere quanto segue. Il processo *A* legge *in* e ne memorizza il valore, 7, in una variabile locale chiamata *primo_slot_libero*; proprio in quell'istante arriva un'interruzione dal clock, quindi la CPU decide che il processo *A* è rimasto in esecuzione per abbastanza tempo e passa al processo *B*, che legge *in* e ottiene 7. Anche *B* memorizza il valore nella sua variabile *primo_slot_libero*: a questo istante, entrambi i processi pensano che il primo slot disponibile sia 7.

Il processo *B* continua la sua esecuzione, memorizza il nome del suo file nell'elemento 7 e aggiorna *in* a 8, poi passa a fare altre cose.

Prima o poi, il processo *A* va di nuovo in esecuzione, dal punto esatto in cui era stato interrotto, accede a *primo_slot_libero*, vi trova 7, e scrive il nome del suo file nell'elemento 7, cancellando il nome che il processo *B* vi aveva appena scritto; poi calcola *primo_slot_libero* + 1, che è 8, e assegna 8 a *in*. Ora la spooler directory è internamente consistente, e quindi il demone della stampante non noterà nulla di sbagliato, ma il processo *B* non otterrà mai niente in uscita, e l'utente *B* bighellonerà intorno alla stanza della stampante per anni, sperando malinconicamente che compaiano dei dati in uscita che non arriveranno mai. Situazioni come queste, nelle quali due o più processi stanno leggendo o scrivendo un qualche dato condiviso ed il risultato finale dipende dall'ordine in cui vengono eseguiti i processi, vengono dette **race condition**, ovvero **corse critiche**. Mettere a punto programmi che contengono corse critiche non è per niente piacevole. I risultati della maggior parte dei test vanno bene, ma una volta ogni tanto accade qualcosa di misterioso ed inspiegabile.

2.3.2 Sezioni critiche

Come si possono evitare le corse critiche? La chiave per prevenire i guai, qui come in molte altre situazioni che coinvolgono memoria condivisa, file condivisi e qualsiasi altra cosa condivisa, è riuscire a trovare qualche modo per impedire che più di un processo alla volta legga o scriva contemporaneamente i dati condivisi. In altre parole, abbiamo bisogno di **mutua esclusione** (*mutual exclusion*), un qualche modo per assicurarsi che se un processo sta usando una variabile o un file condivisi, gli altri processi saranno impossibilitati a fare la stessa cosa. La difficoltà appena descritta sorgeva perché il processo *B* cominciava ad usare le variabili condivise prima che il processo *A* avesse a sua volta finito di usarle. La scelta di primitive appropriate che permettano di realizzare la mutua esclusione è uno dei maggiori problemi che vanno affrontati in fase di progetto di ogni sistema operativo e sarà uno degli argomenti che esamineremo in dettaglio nelle prossime sezioni.

Il problema di evitare le corse critiche può anche essere formulato in maniera astratta. Per un certo tempo, un generico processo è occupato nello svolgimento di calcoli interni o altre operazioni che non portano a corse critiche. Qualche volta, però, un processo può accedere a memoria o file condivisi, o svolgere azioni che possono portare a corse critiche. La parte del programma in cui questo accede alla memoria condivisa viene chiamata **regione critica** (*critical region*) o **sezione critica** (*critical section*). Le corse critiche potrebbero essere evitate se potessimo organizzare le cose in maniera tale che due programmi non possano mai essere nelle loro sezioni critiche nello stesso istante.

Sebbene questo requisito eviti le corse critiche, esso non è sufficiente per avere processi che cooperino fra di loro e che usino i dati condivisi in maniera efficiente. Per avere una buona soluzione, dobbiamo soddisfare quattro condizioni:

1. Due processi non devono mai trovarsi contemporaneamente all'interno delle loro sezioni critiche.
2. Non si deve fare alcuna ipotesi sulle velocità e sul numero delle CPU.
3. Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi.

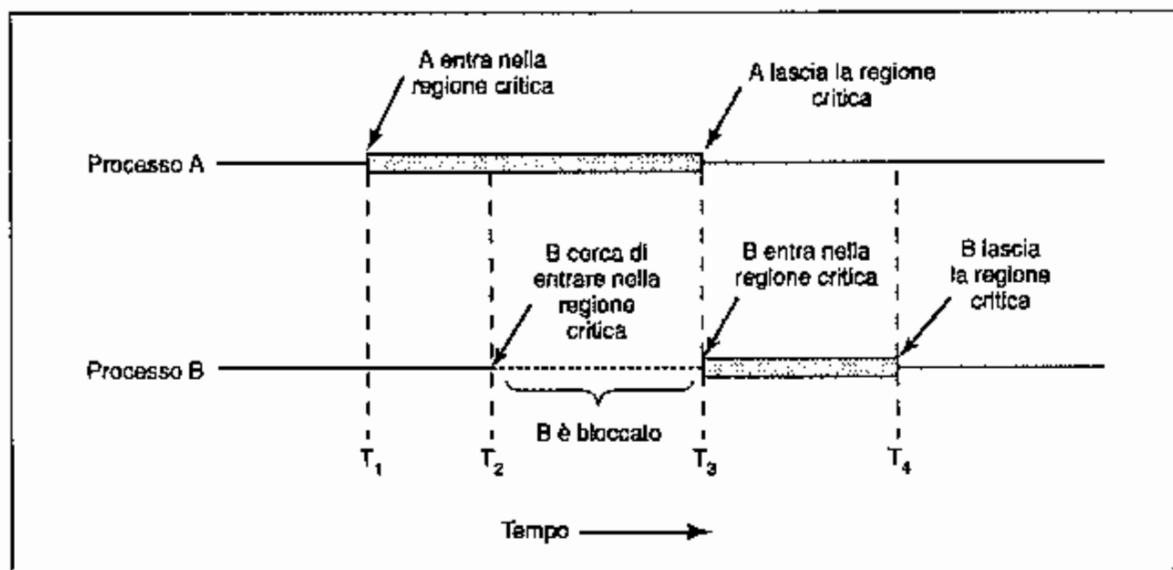


Figura 2.19 Mutua esclusione usando le sezioni critiche.

4. Nessun processo deve aspettare indefinitamente per poter entrare in una sua sezione critica.

In senso astratto, il comportamento che vogliamo è mostrato in Figura 2.19, nella quale il processo A entra nella sua regione critica al tempo T_1 ; poco dopo, al tempo T_2 , il processo B tenta di entrare nella sua regione critica ma fallisce, perché un altro processo è già nella sua regione critica e noi ne permettiamo solo uno per volta. Di conseguenza, B viene temporaneamente sospeso, finché, al tempo T_3 , A lascia la sua sezione critica, permettendo a B di entrare nella sua immediatamente. Alla fine, B esce (al tempo T_4) e ritorniamo alla situazione di partenza, nella quale non ci sono processi nelle sezioni critiche.

2.3.3 Mutua esclusione con attesa attiva

In questa sezione esamineremo varie proposte per ottenere la mutua esclusione, in modo che quando un processo è impegnato ad aggiornare la memoria condivisa nella sua regione critica, nessun altro processo entrerà nella sua regione critica, provocando guai.

Disabilitazione delle interruzioni

La soluzione più semplice è di permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua regione critica, e di riabilitarle non appena ne esce. Con le interruzioni disabilitate, non può arrivare nessuna interruzione dal clock, e siccome la CPU passa da un processo all'altro solo come risultato di un'interruzione da clock o da altro periferico, con le interruzioni disabilitate, la CPU non può venire assegnata ad un altro processo. Così, una volta che il processo ha disabilitato le interruzioni, può esaminare e modificare la memoria condivisa senza paura che un altro processo possa intervenire.

Questo approccio non è molto attraente, in generale, dal momento che non è saggio dare ad un processo utente il potere di disabilitare le interruzioni: e se un processo utente le disabilitasse e poi non le riabilitasse mai più? Potrebbe essere la fine del sistema. In più, in un elaboratore multiprocessore, con due o più CPU, la disabilitazione delle interruzioni avrebbe effetto sulla CPU che esegue l'istruzione `disable`, mentre le altre continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa.

D'altra parte, spesso è conveniente che lo stesso kernel disabiliti le interruzioni per poche istruzioni, mentre sta aggiornando variabili o liste. Se si verifica un'interruzione mentre la lista dei processi pronti, per esempio, è in uno stato inconsistente, si potrebbero verificare corse critiche. La conclusione è che disabilitare le interruzioni è una tecnica spesso utile nel kernel, ma non è appropriata come meccanismo di mutua esclusione fra processi utente.

Variabili di lock

Come secondo tentativo, vediamo una soluzione di tipo software. Supponiamo di avere una singola variabile condivisa (di lock), con valore iniziale 0. Quando un processo vuole entrare nella sua regione critica, controlla prima la variabile di lock: se vale 0, il processo la mette a 1 e entra nella sua regione critica, mentre se la variabile di lock è già posta ad 1, il processo aspetta semplicemente che divenga 0. In questo modo, uno 0 significa che non c'è alcun processo in regione critica, mentre un 1 significa che qualche processo sta nella sua regione critica. Sfortunatamente, questa idea presenta lo stesso incon-

veriente fatale che abbiamo già visto nel caso della directory di spool. Supponiamo che un processo legga la variabile di lock e veda che contiene 0, ma prima che possa metterla a 1, viene schedulato un altro processo, che va in esecuzione e setta la variabile di lock a 1. Quando il primo processo ritorna in esecuzione, impostarà a sua volta la variabile a 1 ed i due processi saranno contemporaneamente nelle loro sezioni critiche.

Ora si potrebbe pensare che il problema si risolva leggendo il valore della variabile di lock e poi controllandolo di nuovo appena prima di scrivere nella variabile, ma questo in realtà non funziona; infatti si verifica una corsa critica se il secondo processo modifica la variabile di lock subito dopo che il primo processo ha effettuato la seconda lettura.

Alternanza stretta

Un terzo approccio al problema della mutua esclusione è quello presentato nella Figura 2.20. Questo frammento di programma, come tutti gli altri presentati nel libro, è scritto in C. Il C è stato scelto perché i sistemi operativi reali sono praticamente sempre scritti in C (o, ogni tanto, in C++) ma quasi mai in linguaggi come Java, Modula 3 o Pascal. Il C è potente, efficiente e prevedibile, caratteristiche adatte per chi scrive sistemi operativi. Java, per esempio, non è prevedibile, perché potrebbe esaurire la memoria allocata in un momento critico, e quindi dover chiamare il garbage collector nel momento più inopportuno. Questo, in C, non può accadere, perché in C non esiste la garbage collection. Un confronto quantitativo di C, C++, Java e altri quattro linguaggi è data in (Prechelt, 2000).

In Figura 2.20, la variabile intera *turno*, inizialmente posta a 0, tiene traccia del processo al quale tocca entrare nella sezione critica per esaminare o aggiornare la memoria condivisa. Inizialmente, il processo 0 legge *turno*, la trova a 0 ed entra nella propria regione critica; anche il processo 1 trova *turno* a 0 e di conseguenza entra in un piccolo ciclo, testando in continuazione *turno* per vedere se sia stata posta a 1. Il testare continuamente una variabile, in attesa che assuma un certo valore, viene detto attesa attiva (*busy waiting*), che di norma dovrebbe essere evitata, dal momento che porta ad uno spreco del tempo di CPU; essa viene usata solo in quei casi in cui ci si aspetta che il tempo di attesa sia breve. Un lock che usa l'attesa attiva è detto *spin lock*.

Quando il processo 0 lascia la regione critica, mette *turno* a 1 per permettere al processo 1 di entrare nella propria sezione critica. Supponiamo che il processo 1 termini velocemente la propria sezione critica, così che entrambi i processi siano fuori dalla propria sezione critica, con la variabile *turno* a 0. Ora, il processo 0 esegue rapidamente l'intero ciclo, uscendo dalla sua sezione critica e ponendo *turno* a 1; a questo punto, *turno* è a 1 e entrambi i processi sono in esecuzione nelle loro sezioni non critiche.

Improvvisamente, il processo 0 termina la sua sezione non critica e torna indietro

<pre> while (TRUE) { while (turno != 0) /* ciclo */ ; regione_critica(); turno = 1; regione_non_critica(); } </pre> <p style="text-align: center;">(a)</p>	<pre> while (TRUE) { while (turno != 1) /* ciclo */ ; regione_critica(); turno = 0; regione_non_critica(); } </pre> <p style="text-align: center;">(b)</p>
--	--

Figura 2.20 Una soluzione proposta al problema delle regioni critiche. (a) Il processo 0. (b) Il processo 1. In entrambi i casi, notate bene il punto e virgola (;) che conclude l'istruzione *while*.

all'inizio del ciclo, ma sfortunatamente, ora non può entrare nella sezione critica, perché *turno* contiene 1 e il processo 1 è ancora occupato con la sua sezione non critica: rimane nel suo ciclo while finché il processo 1 pone *turno* a 0. In altre parole, stabilire dei turni non è una buona idea quando un processo risulti molto più lento dell'altro.

Questa situazione viola la condizione 3 enunciata prima: il processo 0 è bloccato da un processo che non è nella propria sezione critica. Ritornando all'esempio della directory di spool discusso precedentemente, se noi ora associamo la regione critica allo scrivere e leggere la directory di spool, al processo 0 non verrebbe permesso di stampare un altro file perché il processo 1 sta facendo qualcos'altro.

In effetti, questa soluzione richiede che i due processi si alternino strettamente nell'entrare nelle sezioni critiche, ad esempio per fare lo spooling dei file, e nessuno dei due processi può mandare due file in spool in una volta sola. Mentre, da una parte, questo algoritmo permette di evitare qualsiasi corsa critica, dall'altra non rappresenta un serio candidato alla soluzione del problema, perché viola la condizione 3.

La soluzione di Peterson

Combinando l'idea dei turni con l'idea delle variabili di lock e di warning (avvertimento), un matematico olandese, T. Dekker, fu il primo a trovare una soluzione software al problema della mutua esclusione che non richiedesse l'alternanza stretta. Per una discussione dell'algoritmo di Dekker, si veda Dijkstra, 1965.

Nel 1981, G. L. Peterson scoprì un algoritmo molto più semplice per garantire la mutua esclusione, rendendo obsoleto l'algoritmo di Dekker. La Figura 2.21 mostra l'algoritmo di Peterson. Questo algoritmo consta di due procedure scritte in ANSI C, il che significa che bisogna fornire un prototipo per le funzioni definite ed usate. Comunque, per risparmiare spazio, non mostriremo i prototipi in questo e negli esempi che seguono.

Prima di usare le variabili condivise (cioè prima di entrare nella propria regione critica), ciascun processo richiama *entra_nella_regione* con il proprio numero di processo, 0 o 1, come parametro, il che può far sì che il processo debba aspettare, se è il caso, fintanto che risulti sicuro entrare nella regione critica. Dopo aver finito di lavorare sulle variabili

```
#define FALSE 0
#define TRUE 1
#define N 2
                                                /* Numero di processi */

int turno;                                     /* A chi tocca? */
int interessato[N];                            /* Tutti i valori inizialmente 0 o FALSE */

void entra_nella_regione(int processo);        /* processo è 0 o 1 */
{
    int altri;                                    /* Numero dell'altro processo */
    altri = 1 - processo;                         /* L'altro processo */
    interessato[processo] = TRUE;                /* Mostra che sei interessato */
    turno = processo;                            /* Imposta il flag */
    while (turno == processo && interessato[altri] == TRUE) /* Istruzione nulla */;
}

void lascia_la_regione(int processo);           /* Processo: chi lascia la regione critica */
{
    interessato[processo] = FALSE;               /* Uscita dalla regione critica */
}
```

Figura 2.21 La soluzione di Peterson per ottenere la mutua esclusione.

condivise, il processo chiama *lascia_la_region*, per indicare che ha finito e per permettere ad un altro processo di entrare, qualora lo desideri.

Vediamo come lavora questa soluzione: inizialmente, nessun processo è nella sezione critica, poi il processo 0 chiama *entra_nella_region*, indica il suo interesse impostando il proprio elemento nel vettore e mette *turno* a 0: dal momento che il processo 1 non è interessato, la *entra_nella_region* termina immediatamente. Se il processo 1 chiamasse ora la *entra_nella_region*, rimarrebbe in attesa all'interno della procedura fino a che *interessato[0]* non diventa FALSE, evento che si può verificare solo quando il processo 0 chiama la *lascia_la_region* per uscire dalla regione critica.

Ora, consideriamo il caso in cui entrambi i processi chiamano la *entra_nella_region* quasi contemporaneamente: entrambi memorizzeranno il proprio numero di processo nella variabile *turno*, e qualunque cosa venga memorizzata, il secondo valore memorizzato è quello che conta, mentre il primo viene sovrascritto e perso. Supponete che il processo 1 scriva per ultimo nella variabile *turno*, così che essa valga 1; quando entrambi i processi arrivano all'istruzione *while*, il processo 0 non la esegue neanche una volta, ed entra direttamente nella sezione critica, mentre il processo 1 cicla e non entra nella sezione critica, finché il processo 0 non esce.

L'istruzione TSL

Occupiamoci ora di una proposta che richiede un piccolo aiuto da parte dell'hardware: molti calcolatori, in particolare quelli progettati pensando a processori multipli, hanno un'istruzione

TSL RX,LOCK

(Test and Set Lock, verifica e impone il blocco) che funziona come segue: mette il contenuto della parola di memoria *lock* nel registro *RX*, e poi memorizza un valore diverso da zero all'indirizzo di memoria di *lock*. Le operazioni di lettura e memorizzazione della parola sono garantite indivisibili: nessun altro processore può accedere alla parola finché l'istruzione non è finita. La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire alle altre CPU di accedere alla memoria finché non ha finito.

Per usare l'istruzione TSL, useremo una variabile condivisa, *lock*, per coordinare l'accesso alla memoria condivisa. Quando *lock* è a 0, qualunque processo può metterla a 1 usando l'istruzione TSL e poi leggere o scrivere nella memoria condivisa; quando ha finito, il processo mette *lock* di nuovo a 0 utilizzando una normale istruzione move.

Come può essere usata questa istruzione per impedire che due processi entrino contemporaneamente nelle loro regioni critiche? La soluzione è riportata in Figura 2.22, in cui viene mostrata una procedura di quattro istruzioni scritta in un linguaggio assembler fittizio (ma tipico). La prima istruzione copia il vecchio valore di *lock* su un registro e poi mette *lock* a 1, dopodiché il vecchio valore di *lock* viene confrontato con 0. Se non è uguale a zero, il blocco era già stato impostato, quindi il programma, semplicemente, torna all'inizio e controlla nuovamente il valore. Prima o poi il valore diventerà 0 (quando il processo che si trova correntemente nella sua sezione critica la concluderà) e la procedura terminerà, con il blocco impostato. Rimettere a zero il blocco è semplice: il programma memorizza semplicemente 0 in *lock*; non sono necessarie istruzioni speciali.

Una soluzione al problema della sezione critica è ora immediata. Prima di entrare nella propria sezione critica, un processo chiama *entra_nella_region*, che fa attesa attiva finché il blocco non è libero, poi acquisisce il controllo e termina; dopo la sezione critica il processo chiama *lascia_la_region*, che memorizza uno 0 in *lock*. Come in tutte le soluzio-

```

entra_nella_regione:
    TSL REGISTRO,LOCK           | Copia lock nel registro e imposta lock a 1
    CMP REGISTRO,#0            | lock è 0?
    JNE entra_nella_regione   | Se non è 0, lock è stata impostata, quindi cicla
    RET | Ritorna al chiamante, che entra nella regione critica

lascia_la_regione:
    MOVE LOCK,#0                | Memorizza uno 0 in lock
    RET | Ritorna al chiamante

```

Figura 2.22 Entrare e uscire dalla regione critica usando l'istruzione TSL.

ni basate sulle regioni critiche, affinché il metodo funzioni i processi devono chiamare *entra_nella_regione* e *lascia_la_regione* al momento giusto: se un processo bara, la mutua esclusione fallirà.

2.3.4 Sospensione e risveglio

Sia la soluzione di Peterson, sia quella che usa TSL sono corrette, ma entrambe hanno il difetto di richiedere attesa attiva. Queste soluzioni fanno essenzialmente una cosa: quando un processo vuole entrare in una sezione critica, esegue un controllo per vedere se l'ingresso nella sezione è possibile. Se non lo è, il processo si blocca nell'esecuzione di un ciclo molto stretto, fino a che l'ingresso non è ammesso.

Questo comportamento non solo porta ad uno spreco di tempo della CPU, ma può anche avere effetti inaspettati. Consideriamo un calcolatore con due processi, *H*, ad alta priorità, e *L*, a bassa priorità; le regole di schedulazione sono tali che *H* viene mandato in esecuzione non appena si trova nello stato di pronto. Ad un certo istante, mentre *L* si trova nella propria sezione critica, *H* passa nello stato di pronto (per esempio, si completa un'operazione di ingresso/uscita). *H* comincia quindi l'attesa attiva, ma poiché *L* non viene mai scelto quando *H* è in esecuzione, *L* non ha mai la possibilità di lasciare la sezione critica, così *H* cicla all'infinito. Questa situazione viene chiamata a volte **problema dell'inversione delle priorità** (*priority inversion problem*).

Vediamo adesso alcune primitive di comunicazione fra processi che li bloccano, anziché sprecare tempo di CPU, quando non è loro permesso di entrare nelle loro sezioni critiche. Una delle coppie di primitive più semplici è formata dalla *sleep* e dalla *wakeup*: *sleep* è una chiamata di sistema che provoca il blocco del processo chiamante, cioè, che lo sospende finché un altro processo non lo risveglia. La chiamata *wakeup* ha un parametro, che rappresenta il processo che deve essere risvegliato. In alternativa, sia la *sleep* sia la *wakeup* hanno un parametro, un indirizzo di memoria che serve ad accoppiare le *sleep* con le *wakeup*.

Il problema del produttore-consumatore

Come esempio del modo in cui si possono usare queste primitive, consideriamo il problema del **produttore-consumatore** (*producer-consumer problem*), conosciuto anche come il problema del **buffer a capacità limitata** (*bounded-buffer problem*). Due processi condividono un unico buffer di dimensioni fisse: uno di loro, il **produttore**, deposita informazioni nel buffer, mentre l'altro, detto **consumatore**, le preleva. (È anche possibile

```

#define N 100                                /* Numero di posti nel buffer */
int cont = 0;                               /* Numero di elementi nel buffer */

void produttore(void)
{
    int elemento;

    while (TRUE) {                         /* Ripeti per sempre */
        elemento = produci_elemento();      /* Genera il prossimo elemento */
        if (cont == N) sleep();             /* Se il buffer è pieno, sospendi */
        inserisci_elemento(elemento);     /* Metti l'elemento nel buffer */
        cont = cont + 1;                  /* Incrementa numero elementi nel buffer */
        if (cont == 1) wakeup(consumatore); /* Il buffer era vuoto? */
    }
}

void consumatore(void)
{
    int elemento;

    while (TRUE) {                         /* Ripeti per sempre */
        if (cont == 0) sleep();             /* Se il buffer è vuoto, sospendi */
        elemento = estrai_elemento();     /* Prendi un elemento dal buffer */
        cont = cont - 1;                 /* Decrementa numero elementi nel buffer */
        if (cont == N - 1) wakeup(cont);   /* Il buffer era pieno */
        consuma_elemento(elemento);      /* Stampa l'elemento */
    }
}

```

Figura 2.23 Il problema del produttore-consumatore con una corsa critica fatale.

generalizzare il problema per avere m produttori e n consumatori, ma considereremo solo il caso di un produttore e un consumatore, perché questo presupposto semplifica le soluzioni.). I problemi nascono quando il produttore vuole depositare un nuovo elemento nel buffer, ma questo è già pieno. La soluzione è quella di sospendere il produttore, per risvegliarlo poi quando il consumatore avrà rimosso uno o più elementi. In maniera analoga, se il consumatore vuole rimuovere un elemento da un buffer vuoto, viene sospeso fino a che il produttore non deposita qualche cosa nel buffer e lo risveglia.

Questo approccio sembra abbastanza semplice, ma porta agli stessi problemi di corse critiche che abbiamo già visto nell'esempio della directory di spool. Per tener conto del numero di elementi presenti nel buffer abbiamo bisogno di una variabile, $cont$. Se il numero massimo di elementi che il buffer può contenere è N , il codice del produttore deve per prima cosa controllare se $cont$ sia uguale ad N ; se lo è, il produttore si sospende, altrimenti il produttore depositerà l'elemento ed incrementerà $cont$.

Il codice del consumatore sarà dello stesso tipo: per prima cosa controllerà se $cont$ vale 0: se sì, si sosponderà, altrimenti rimuoverà un elemento dal buffer. Ciascuno dei processi controllerà inoltre se l'altro debba essere svegliato e, in questo caso, lo risveglierà. Il codice per il produttore e per il consumatore è presentato in Figura 2.23.

Per esprimere chiamate di sistema come `sleep` e `wakeup` in C, le mostreremo come

chiamate a procedure di libreria. Esse non fanno parte della libreria standard del C, ma, presumibilmente, saranno disponibili in qualunque sistema che abbia veramente queste chiamate di sistema. Le procedure *inserisci_elemento* e *estrai_elemento*, che non sono descritte, gestiscono i dettagli dell'inserzione e dell'estrazione degli elementi nel buffer.

Adesso torniamo alla corsa critica: essa può avvenire perché gli accessi alla variabile *cont* non sono in alcun modo regolamentati. Potrebbe accadere la seguente situazione: il buffer è vuoto ed il consumatore ha appena letto *cont* per vedere se vale 0; in quel momento, lo schedulatore decide di sospendere temporaneamente il consumatore e manda in esecuzione il produttore. Il produttore inserisce un elemento nel buffer, incrementa *cont* e verifica che vale 1; rendendosi conto che *cont* era poco prima a 0 e che quindi il consumatore era sospeso, il produttore chiama la *wakeup*, per risvegliare il consumatore.

Sfortunatamente, il consumatore non è ancora logicamente sospeso, quindi il segnale di sveglia va perso. Quando il consumatore torna in esecuzione, verificherà il valore di *cont* che aveva precedentemente letto, scoprirà che è 0 e si sospenderà. Prima o poi, il produttore riempirà il buffer ed anch'esso si sospenderà; entrambi resteranno sospesi per sempre.

L'essenza del problema, qui, è che una sveglia spedita ad un processo che non è (ancora) sospeso, va persa; se non andasse persa, andrebbe tutto bene. Un rimedio veloce è quello di modificare le regole per aggiungere al quadro un bit di attesa della sveglia (*wakeup waiting bit*). Quando viene spedita una sveglia ad un processo che è già sveglio, questo bit viene messo a 1. Più tardi, quando il processo cercherà di sospendersi, se il bit di attesa della sveglia è a 1 verrà messo a 0, ma il processo non si sospenderà. Il bit di attesa della sveglia è in realtà un salvadanaio per i segnali di sveglia.

Mentre il bit di attesa della sveglia risolve il problema in questo semplice esempio, è facile costruire esempi con tre o più processi, nei quali un bit di attesa della sveglia risulta insufficiente. Potremmo introdurre un altro rimedio ed aggiungere un secondo bit, o forse 8 o 32 bit, ma di principio il problema rimane.

2.3.5 I semafori

Questa era la situazione nel 1965, quando E. W. Dijkstra (1965) suggerì di usare una variabile intera per contare il numero di sveglie salvate per l'uso futuro. Nella sua proposta venne introdotto un nuovo tipo di variabile, chiamato **semaforo**: esso può avere valore 0, il che indica che non è stata salvata alcuna sveglia, o un qualche valore positivo, se una o più sveglie sono già pervenute.

Dijkstra propose due operazioni, *down* e *up* (generalizzazioni di *sleep* e *wakeup*, rispettivamente). L'operazione *down* su un semaforo controlla che il valore sia maggiore di 0; se è così, decrementa il valore (cioè consuma una delle sveglie che erano state salvate) e semplicemente continua, mentre se il valore è 0, il processo viene sospeso, per il momento senza completare la *down*. Controllare il valore, cambiarlo ed eventualmente sosprendersi sono tutte operazioni fatte in un'unica, indivisibile **azione atomica**. Viene garantito che, una volta che un'operazione su un semaforo è cominciata, nessun altro processo possa accedere al semaforo fino a che l'operazione non è completata o bloccata. Questa atomicità è assolutamente essenziale per risolvere i problemi di sincronizzazione e per evitare corse critiche.

L'operazione *up* incrementa il valore del semaforo su cui viene invocata. Se uno o più processi erano sospesi su quel semaforo, incapaci di completare una precedente operazione *down*, uno di essi viene scelto dal sistema (per esempio a caso) per permettergli di completare la propria *down*. Così, dopo una *up* su un semaforo che ha processi sospesi,

il semaforo sarà ancora 0, ma ci sarà un processo in meno tra quelli sospesi. Anche l'operazione di incremento del semaforo e di risveglio di uno dei processi è indivisibile, e nessun processo si blocca facendo una up, così come nessun processo si blocca facendo una wakeup del modello precedente.

Per inciso, nel suo lavoro originale Dijkstra aveva usato i nomi P e V, al posto di down e up, rispettivamente, ma dal momento che questi non hanno alcun significato mnemonico per chi non parla olandese (e solo un significato marginale per quelli che lo parlano) noi useremo i termini down e up, introdotti per la prima volta nell'Algol 68.

Risolvere il problema del produttore-consumatore con i semafori

I semafori risolvono il problema delle sveglie perdute, come mostrato in Figura 2.24 ed è essenziale che essi siano implementati in maniera indivisibile. Normalmente up e down sono implementate come chiamate di sistema, con il sistema operativo che disabilita per un attimo tutte le interruzioni mentre sta verificando il valore del semaforo, aggiornandolo e sospendendo il processo, se necessario; dal momento che tutte queste azioni durano solo poche istruzioni, disabilitare le interruzioni non fa alcun danno. Nel caso si usino CPU multiple, ciascun semaforo deve essere protetto da una variabile di lock, e si deve usare l'istruzione TSL per garantire che solo una CPU alla volta esamini il semaforo. È importante capire che l'uso della TSL per impedire a più CPU di accedere contemporaneamente al semaforo è abbastanza diverso dall'attesa attiva fatta dal produttore o dal consumatore in attesa che l'altro vuoti o riempia il buffer: l'operazione sul semaforo richiederà soltanto pochi microsecondi, mentre il produttore o il consumatore potrebbero impiegare un tempo arbitrariamente lungo.

Questa soluzione usa tre semafori: uno chiamato pieni per contare il numero di elementi che sono occupati, uno chiamato vuoti per contare il numero di elementi vuoti, ed uno chiamato mutex per garantire che il produttore ed il consumatore non accedano al buffer contemporaneamente. pieni è inizialmente posto a 0, vuoti è inizialmente posto uguale al numero degli elementi del buffer e mutex è inizialmente posto a 1. I semafori che vengono inizializzati a 1 e usati da due o più processi per assicurarsi che di volta in volta solo uno di essi possa entrare nella propria regione critica, sono chiamati semafori binari. Se ogni processo chiama una down subito prima di entrare nella propria regione critica ed una up subito dopo esserne uscito, è garantita la mutua esclusione.

Ora che abbiamo a disposizione una buona primitiva per la comunicazione fra processi, torniamo indietro ed esaminiamo di nuovo la sequenza di interruzione di Figura 2.5. In un sistema che usa semafori, il modo naturale per mascherare le interruzioni è di avere un semaforo, inizializzato a 0, associato ad ogni dispositivo di ingresso/uscita. Subito dopo aver fatto partire un dispositivo di ingresso/uscita, il processo gestore chiama una down sul semaforo associato, bloccandosi immediatamente. Quando arriva l'interruzione, il gestore dell'interruzione chiama una up sul semaforo associato, che rende il processo interessato pronto ad andare di nuovo in esecuzione. In questo modello, il passo 5 in Figura 2.5 consiste nel chiamare una up sul semaforo del dispositivo, in modo che al passo 6 lo scheduler possa mandare in esecuzione il gestore del dispositivo; naturalmente, se ora diversi processi sono pronti, lo scheduler può scegliere di mandare in esecuzione un processo più importante. Ci occuperemo di come viene fatta la schedulazione più avanti in questo stesso capitolo.

Nell'esempio di Figura 2.24, abbiamo effettivamente usato i semafori in due modi diversi, e la differenza è abbastanza importante da dover essere sottolineata. Il semaforo mutex è utilizzato per la mutua esclusione, per garantire che solo un processo alla volta leggerà o scriverà il buffer e le variabili associate. Questa mutua esclusione è necessaria

```

#define N 100                                /* Numero delle posizioni del buffer */
typedef int semaforo;                      /* I semafori sono un tipo di interi */
semaforo mutex = 1;                         /* Controlla l'accesso alla regione critica */
semaforo vuoti = N;                         /* Conta le posizioni vuote nel buffer */
semaforo pieni = 0;                         /* Conta le posizioni piene nel buffer */

void produttore(void)
{
    int item;

    while (TRUE) {
        item = produci_elemento( );
        down(&vuoti);                      /* Genera qualcosa da inserire nel buffer */
        down(&mutex);                     /* Decrementa contatore posizioni vuote */
        inserisci_elemento(item);          /* Entra nella regione critica */
        up(&mutex);                      /* Metti il nuovo elemento nel buffer */
        up(&vuoti);                       /* Lascia la regione critica */
        up(&pieni);                      /* Incrementa contatore posizioni piene */
    }
}

void consumatore(void)
{
    int item;

    while (TRUE) {
        down(&pieni);                    /* Ciclo infinito */
        down(&mutex);                     /* Decrementa contatore posizioni piene */
        item = estrai_elemento( );        /* Entra nella regione critica */
        up(&mutex);                      /* Prendi un elemento dal buffer */
        up(&vuoti);                      /* Abbandona la regione critica */
        up(&vuoti);                      /* Incrementa contatore posizioni vuote */
        consuma_elemento(item);          /* Fai qualcosa con l'elemento */
    }
}

```

Figura 2.24 Il problema del produttore consumatore con i semafori.

per impedire il caos. Studieremo ulteriormente la mutua esclusione e come ottenerla nella prossima sezione.

L'altro uso dei semafori è per la sincronizzazione. I semafori *pieni* e *vuoti* sono necessari per garantire che certe sequenze di eventi si verifichino o no in un certo ordine. In questo caso assicurano che il produttore si fermi quando il buffer è pieno e che il consumatore si fermi quando esso è vuoto. Questo uso è diverso dalla mutua esclusione.

2.3.6 I mutex

Quando la capacità di contare dei semafori non serve, ne viene qualche volta utilizzata una versione semplificata, detta mutex. I mutex vanno bene solo per gestire la mutua esclusione di qualche risorsa o di qualche pezzo di codice condivisi; sono facili ed efficienti da implementare, il che li rende particolarmente utili nei thread implementati nello spazio utente.

Un mutex è una variabile che può essere in uno di due stati: bloccato non o bloccato. Di conseguenza, basta solo un bit per rappresentarlo, ma in pratica viene spesso usato un

```

mutex_lock:
    TSL REGISTRO,MUTEX      | Copia mutex in un registro e imposta mutex a 1
    CMP REGISTRO,#0          | Mutex era zero (0)?
    JZE ok                  | Se era zero, mutex era non bloccato, quindi termina
    CALL thread_yield        | Mutex è occupato; schedula un altro thread
    JMP mutex_lock           | Riprova più tardi
ok: RET | Ritorna al chiamante, che entra nella regione critica

mutex_unlock:
    MOVE MUXTEX,#0           | Memorizza 0 in mutex
    RET | Ritorna al chiamante

```

Figura 2.25 Implementazione di *mutex_lock* e *mutex_unlock*.

intero, con il valore 0 che indica lo stato di non bloccato e tutti gli altri valori che indicano bloccato. Con i mutex vengono usate due procedure: quando un thread (o un processo) deve accedere ad una regione critica, richiama *mutex_lock* e se il mutex si trova correntemente nello stato di non bloccato (il che indica che la regione critica è disponibile), la chiamata ha successo e il thread chiamante è libero di entrare nella regione critica.

D'altro canto, se il mutex è già bloccato, il thread viene bloccato finché il thread nella regione critica ha finito e chiama *mutex_unlock*. Se più thread sono bloccati sul mutex, uno di questi viene scelto a caso e questo può acquisire la risorsa.

Poiché i mutex sono così semplici, possono essere implementati facilmente nello spazio utente, se è disponibile un'istruzione TSL. Il codice per *mutex_lock* e *mutex_unlock* da usare con i thread di livello utente è mostrato in Figura 2.25.

Il codice di *mutex_lock* è simile al codice di *entra_nella_regione* di Figura 2.22, ma con una differenza importante: quando *entra_nella_regione* fallisce l'entrata nella regione critica, controlla il blocco ripetutamente (attesa attiva). Prima o poi, il clock scade e un altro processo viene selezionato per l'esecuzione, quindi il processo che detiene il blocco andrà in esecuzione e lo rilascerà.

Con i thread, la situazione è diversa, perché non c'è clock per fermare i thread che sono stati in esecuzione troppo a lungo. Di conseguenza, un thread che prova ad acquisire un blocco tramite attesa attiva starà in ciclo per sempre e non acquisirà mai il blocco, perché non permette mai ad altri thread di essere in esecuzione e di rilasciarlo.

Questo è il motivo per cui c'è differenza fra *entra_nella_regione* e *mutex_lock*. Quando il thread fallisce nell'acquisizione del blocco, richiama *thread_yield* per lasciare la CPU a un altro thread, quindi non c'è attesa attiva; quando tornerà in esecuzione, effettuerà di nuovo il test sul blocco.

Dal momento che *thread_yield* è solo una chiamata allo scheduler dei thread nello spazio utente, è molto veloce, quindi né *mutex_lock* né *mutex_unlock* richiedono chiamate di sistema. Usandole, thread di livello utente possono sincronizzarsi completamente nello spazio utente, usando procedure che richiedono solo una manciata di istruzioni.

Il sistema dei mutex che è stato descritto sopra è un insieme di chiamate essenziali. Come in tutto il software, c'è sempre richiesta di nuove caratteristiche, e le primitive di sincronizzazione non fanno eccezione: ad esempio, qualche volta le librerie dei thread offrono una chiamata *mutex_trylock* che o acquisisce il blocco o restituisce un codice che indica l'insuccesso, ma non blocca: tale chiamata dà al thread la flessibilità di decidere cosa fare se ci sono alternative all'attendere semplicemente.

Fino ad ora, c'è un problema su cui abbiamo sorvolato ma che vale la pena di eviden-

ziare. Con i thread nello spazio utente, non c'è problema se molti thread hanno accesso allo stesso mutex, dal momento che tutti i thread operano nello stesso spazio di indirizzamento. Comunque, nella maggior parte delle soluzioni precedenti, come l'algoritmo di Peterson e i semafori, c'è l'ipotesi tacita che processi multipli abbiano accesso ad almeno un po' di memoria condivisa, forse solo una parola, ma qualcosa. Se i processi hanno spazio di indirizzamento separato, come abbiamo continuamente detto, come possono condividere la variabile *turno* dell'algoritmo di Peterson, o i semafori o un buffer comune?

Ci sono due risposte. Primo, alcune delle strutture dati condivise, come i semafori, possono essere memorizzate nel kernel e accedute solo tramite chiamate di sistema e questo approccio elimina il problema. Secondo, la maggior parte dei sistemi operativi moderni (compresi UNIX e Windows) fornisce un modo attraverso il quale i processi possono condividere porzioni del loro spazio di indirizzamento con altri processi, così i buffer e altre strutture dati possono essere condivise. Nel caso peggiore, quando non è possibile nient'altro, può essere usato un file condiviso.

Se due o più processi condividono la maggior parte o tutto il loro spazio di indirizzamento, la distinzione fra processi e thread diventa indistinta, ma è nondimeno presente. Due processi che condividono lo spazio di indirizzamento possono avere differenti file aperti, timer di allarme e altre proprietà dei processi, mentre i thread in un singolo processo condividono queste cose, ed è sempre vero che più processi che condividono uno spazio comune non hanno mai l'efficienza di thread a livello utente, dal momento che il kernel è profondamente coinvolto nella loro gestione.

2.3.7 I monitor

Con i semafori la comunicazione fra processi sembra semplice, no? Scordatevelo. Osservate attentamente l'ordine delle down prima di immettere o di riunire elementi dal buffer in Figura 2.24 e supponete di scambiare l'ordine di due down nel codice del produttore, così che *mutex* sia decrementato prima di vuoti invece che dopo di esso. Se il buffer fosse completamente pieno, il produttore si bloccherebbe, con *mutex* a 0 e, di conseguenza, la prossima volta che il consumatore cercasse di accedere al buffer, chiamerebbe una down su *mutex*, ora a 0, ed anch'esso si bloccherebbe; entrambi i processi rimarrebbero bloccati per sempre e non verrebbe svolto più altro lavoro. Questa situazione sfavorevole è chiamata **stallo** (deadlock); la studieremo in dettaglio nel Capitolo 3.

Questo problema è stato evidenziato per mostrare con quanta attenzione si debbano usare i semafori: un piccolo errore e si inceppa tutto quanto. È come programmare in linguaggio assembler, solo peggio, perché gli errori portano a corse critiche, stalli ed altre forme di comportamento non prevedibili e non riproducibili.

Per rendere più semplice la scrittura di programmi corretti, Hoare (1974) e Brinch Hansen (1975) proposero una primitiva di sincronizzazione a più alto livello, chiamata **monitor**; le loro proposte differiscono leggermente, come descritto di seguito. Un monitor è una collezione di procedure, variabili e strutture dati che vengono raggruppate insieme in un tipo speciale di modulo o package. I processi possono chiamare le procedure di un monitor ogni volta che vogliono, ma non possono accedere direttamente alle strutture dati interne del monitor da procedure dichiarate al di fuori del monitor stesso. La Figura 2.26 illustra un monitor scritto in un linguaggio immaginario, pseudo Pascal.

I monitor possiedono un'importante proprietà che li rende utili per ottenere la mutua esclusione: ad ogni istante, un solo processo può essere attivo in un monitor. I monitor sono un costrutto del linguaggio di programmazione, quindi il compilatore sa che sono speciali e può gestire le chiamate alle procedure del monitor in maniera diversa rispetto alle altre chiamate di procedura. Tipicamente, quando un processo chiama una procedu-

```

monitor esempio
    integer i;
    condition c;

    procedure produttore();
    . .
    .
    ond;

    procedure consumatore();
    . .
    .
    ond;
end monitor;

```

Figura 2.26 Un monitor.

ra di monitor, le prime istruzioni della procedura controlleranno se un qualche altro processo è correntemente attivo dentro il monitor; se è così, il processo chiamante sarà sospeso, finché l'altro processo non avrà lasciato il monitor. Se nessun altro processo sta usando il monitor, il processo chiamante può entrare.

Spetta al compilatore implementare la mutua esclusione sui punti di ingresso del monitor, sebbene un modo comune sia quello di utilizzare un mutex o semaforo binario. Poiché è il compilatore e non il programmatore che predispone la mutua esclusione, è molto meno probabile che qualche cosa vada storta. In ogni caso, colui che scrive il monitor non deve essere a conoscenza di come il compilatore predispone la mutua esclusione; è sufficiente sapere che, trasformando tutte le sezioni critiche in procedure del monitor, due processi qualunque non potranno mai eseguire contemporaneamente le proprie sezioni critiche.

Sebbene i monitor forniscano una maniera semplice per ottenere la mutua esclusione, come abbiamo appena visto, questo non è sufficiente, perché abbiamo anche bisogno di un modo per bloccare i processi quando non possono proseguire. Nel problema del produttore-consumatore è abbastanza facile mettere tutti i controlli di buffer-pieno e buffervuoto in procedure di monitor, ma come dovrebbe bloccarsi il produttore quando trova il buffer pieno?

La soluzione sta nell'introduzione di variabili di tipo condizione (condition variable), con due operazioni associate, *wait* e *signal*; quando una procedura di monitor scopre che non può continuare (ad esempio, il produttore trova il buffer pieno), chiama una *wait* su una qualche variabile di tipo condizione, diciamo *pieni*. Questa azione provoca il blocco del processo chiamante e permette l'ingresso ad un altro processo a cui era stato precedentemente proibito di entrare nel monitor.

Quest'altro processo, ad esempio il consumatore, può svegliare il suo partner sospeso chiamando una *signal* sulla variabile di tipo condizione su cui questo è in attesa. Per evitare di avere contemporaneamente due processi attivi nel monitor, abbiamo bisogno di una regola che stabilisca cosa accade dopo una *signal*. Hoare propose di permettere al processo appena risvegliato di andare in esecuzione, sospendendo l'altro; Brinch Hansen propose di risolvere astutamente il problema imponendo che un processo che chiama una *signal* debba uscire immediatamente dal monitor; in altre parole, un'istruzione *signal* può comparire soltanto come istruzione finale di una procedura di monitor. Noi useremo la proposta di Brinch Hansen perché è più semplice ed è anche più facile da

implementare; se si fa una `signal` su una variabile di tipo condizione su cui sono sospesi diversi processi, soltanto uno di essi, determinato dallo scheduler di sistema, verrà risvegliato.

Detto per inciso, c'è anche una terza soluzione, non proposta né da Hoare né da Brinch Hansen, che è lasciare in esecuzione il processo che ha fatto la `signal` e permettere al processo che sta aspettando di iniziare l'esecuzione solo dopo che il primo processo ha abbandonato il monitor.

Le variabili di tipo condizione non sono contatori, né accumulano segnali per un uso successivo come fanno i semafori, quindi, se viene eseguita una `signal` su una variabile di tipo condizione su cui nessuno è sospeso, il segnale viene perso. In altre parole, la `wait` deve avvenire prima della `signal`; questa regola rende molto più semplice l'implementazione; in pratica ciò non è un problema dal momento che risulta semplice tener traccia dello stato di ciascun processo tramite delle variabili, se è il caso. Un processo che deve eseguire una `signal` può rendersi conto se questa operazione non è necessaria controllando tali variabili.

Uno scheletro della soluzione del problema produttore-consumatore con i monitor è illustrato in Figura 2.27, in un linguaggio immaginario, pseudo Pascal. Il vantaggio del pseudo Pascal qui è che è puro e semplice e segue esattamente il modello di Hoare e Brinch Hansen.

Potreste pensare che le operazioni di `wait` e `signal` siano molto simili alle `sleep` e `wakeup`, che, come abbiamo visto prima, portano a corse critiche fatali. Esse sono molto simili, ma con una sostanziale differenza: `sleep` e `wakeup` fallivano perché, mentre un processo cercava di sospendersi, l'altro cercava di risveglierlo, mentre con i monitor questo non può accadere. La mutua esclusione automatica sulle procedure del monitor garantisce che se, supponiamo, il produttore scopre dentro a una procedura di monitor che il buffer è pieno, sarà in grado di completare l'operazione di `wait` senza doversi preoccupare della possibilità che lo scheduler passi il controllo al consumatore prima che la `wait` sia completata. Al consumatore non verrà proprio permesso di entrare nel monitor finché la `wait` non è finita ed il produttore sia stato marcato come non più eseguibile.

Sebbene lo pseudo Pascal sia un linguaggio immaginario, anche qualche vero linguaggio di programmazione supporta i monitor, sebbene non sempre nella forma progettata da Hoare e Brinch Hansen. Un linguaggio di questo tipo è Java: esso è un linguaggio orientato agli oggetti che supporta thread a livello utente e permette anche ai metodi (procedure) di essere raggruppati in classi. Aggiungendo la parola chiave `synchronized` alla dichiarazione di metodo, Java garantisce che una volta che un thread qualsiasi ha iniziato l'esecuzione di quel metodo, a nessun altro thread sarà permesso di iniziare l'esecuzione di un altro metodo `synchronized` in quella classe.

Una soluzione al problema del produttore consumatore utilizzando i monitor in Java è data in Figura 2.28. La soluzione consiste di quattro classi. La classe esterna, *Produttore-Consumatore*, crea e inizia l'esecuzione di due thread, *p* e *c*; la seconda e la terza classe, *produttore* e *consumatore*, rispettivamente, contengono il codice del produttore e del consumatore; infine, la classe *nostro_monitor*, è il monitor e contiene due thread `synchronized` che vengono utilizzati per inserire veramente e prelevare elementi nel buffer condiviso. Diversamente dagli esempi precedenti, abbiamo infine riportato il codice completo di *inserisci* e *estrai*.

I thread di produttore e consumatore sono funzionalmente identici alle loro controparti di tutti gli esempi precedenti. Il produttore ha un ciclo infinito che genera dati e li mette in un buffer comune; il consumatore ha un ciclo infinito equivalente, in cui preleva dati dal buffer comune e ci gioca.

La parte interessante di questo programma è la classe *nostro_monitor*, che contiene il buffer, le variabili di amministrazione e due metodi sincronizzati: quando il produttore è

attivo in *inserisci*, sa sicuramente che il consumatore non può essere attivo in *estrai*, rendendo sicuro l'aggiornamento delle variabili e del buffer, senza paura di corse critiche. La variabile *cont* tiene traccia di quante elementi sono nel buffer, e può assumere ogni valore da 0 a $N - 1$. La variabile *lo* è l'indice della posizione nel buffer dove il prossimo elemento deve essere preso e, in modo simile, *hi* è l'indice della posizione nel buffer dove deve essere messo il prossimo elemento; si permette che $lo = hi$, che significa che 0 o N elementi sono nel buffer. Il valore di *cont* dice quale dei due casi si è verificato.

In Java i metodi sincronizzati differiscono dai monitor tradizionali in modo sostan-

```

monitor ProduttoreConsumatore
    condition pieni, vuoti;
    integer cont;

    procedure inserisci(item: integer);
    begin
        if cont = N then wait(pieni);
        inserisci_elemento(item);
        cont := cont + 1;
        if cont = 1 then signal(vuoti)
    end;

    function estrai: integer;
    begin
        if cont = 0 then wait(vuoti);
        estrai = estrai_elemento;
        cont := cont - 1;
        if cont = N - 1 then signal(pieni)
    end;

    cont := 0;
end monitor;

procedure produttore;
begin
    while true do
    begin
        item = produci_elemento;
        ProduttoreConsumatore.inserisci(item)
    end
end;

procedure consumatore;
begin
    while true do
    begin
        item = ProduttoreConsumatore.estrai;
        consuma_elemento(item)
    end
end;

```

Figura 2.27 Una bozza del problema del produttore-consumatore con i monitor. Solo una procedura del monitor per volta è attiva. Il buffer ha N elementi.

```

public class ProduttoreConsumatore {
    static finale int N = 100; // Costante che dà la dimensione del buffer
    static produttore p = new produttore( ); // Istanzia un nuovo thread produttore
    static consumatore c = new consumatore( ); // Istanzia un nuovo thread consumatore
    static nostro_monitor mon = new nostro_monitor( ); // Istanzia un nuovo monitor

    public static void main(String args[ ]) {
        p.start( ); // Inizia il thread produttore
        c.start( ); // Inizia il thread consumatore
    }

    static class produttore extends Thread {
        public void run( ) { // Esegue un metodo che contiene codice di thread
            int item;
            while (true) { // Ciclo produttore
                item = produci_elemento( );
                mon.inserisci(item);
            }
        }
        private int produci_elemento( ) { ... } // Produce veramente
    }

    static class consumatore extends Thread {
        public void run( ) { // Esegue un metodo che contiene codice di thread
            int item;
            while (true) { // Ciclo consumatore
                item = mon.estrai( );
                consuma_elemento (item);
            }
        }
        private void consuma_elemento(int item) { ... } // Consuma veramente
    }

    static class nostro_monitor { // Questo è un monitor
        private int buffer[ ] = new int[N];
        private int cont = 0, lo = 0, hi = 0; // Contatori e indici

        public synchronized void inserisci(int val) {
            if (cont == N) vai_a_dormire( ); // Se il buffer è pieno, suspenditi
            buffer [hi] = val; // Inserisce un elemento nel buffer
            hi = (hi + 1) % N; // Posizione in cui mettere il prossimo elemento
            cont = cont + 1; // C'è un nuovo elemento nel buffer, ora
            if (cont == 1) notify( ); // Se il consumatore è sospeso, sveglialo
        }

        public synchronized int estrai( ) {
            int val;
            if (cont == 0) vai_a_dormire( ); // Se il buffer è vuoto, suspenditi
            val = buffer [lo]; // Prendi un elemento dal buffer
            lo = (lo + 1) % N; // Posizione da cui prendere elementi dal buffer
            cont = cont - 1; // Un elemento in meno nel buffer
            if (cont == N - 1) notify( ); // Se il produttore è sospeso, sveglialo
            return val;
        }
        private void vai_a_dormire( ) { try{wait( );} catch(InterruptedException exc) {}}
    }
}

```

Figura 2.28 Una soluzione al problema del produttore-consumatore in Java.

ziale: Java non ha variabili di tipo condizione, ma ha due procedure, *wait* e *notify*, che sono le equivalenti di *sleep* e *wakeup*, eccettuato che, quando vengono usate in metodi sincronizzati, non sono soggette a corse critiche. In teoria, il metodo *wait* può essere interrotto, e la gestione di questo caso è lasciata al codice che lo circonda; poi, Java richiede che la gestione delle eccezioni sia esplicita. Per i nostri scopi, immaginate semplicemente che *vai_a_dormire* sia il modo per sospendersi.

Rendendo automatica la mutua esclusione sulle regioni critiche, i monitor rendono la programmazione parallela molto meno soggetta ad errori rispetto ai semafori, ma anch'essi presentano alcuni inconvenienti. Non è per caso che i nostri due esempi siano scritti in pseudo Pascal e in Java piuttosto che in C, come tutti gli altri esempi in questo libro. Come abbiamo già detto, i monitor sono un concetto dei linguaggi di programmazione; il compilatore deve riconoscerli ed organizzare in un qualche modo la mutua esclusione. C, Pascal e la maggior parte degli altri linguaggi non hanno i monitor, e quindi è irragionevole aspettarsi che i loro compilatori impongano una qualunque regola per la mutua esclusione; infatti, come potrebbe sapere il compilatore quali procedure stanno nei monitor e quali no?

Questi stessi linguaggi non hanno neanche i semafori, ma la loro aggiunta risulta facile: basta aggiungere alla libreria due piccole procedure in linguaggio assembler per chiamare le chiamate di sistema *up* e *down*; i compilatori non hanno nemmeno bisogno di sapere che esistono. Naturalmente, i sistemi operativi devono essere a conoscenza dell'esistenza dei semafori, ma almeno, se avete un sistema operativo basato sui semafori, potete scrivere i programmi utente per questo sistema operativo in C o in C++ (o persino in linguaggio assembly se siete abbastanza masochisti); con i monitor, ci vuole un linguaggio che li sappia gestire.

Un altro problema con i monitor, e anche con i semafori, è che furono progettati per risolvere il problema della mutua esclusione su una o più CPU con accesso a una memoria comune. Mettendo i semafori in memoria condivisa e proteggendoli con istruzioni TSL, possiamo evitare corse critiche. Passando ad un sistema distribuito formato da CPU multiple, ciascuna con la sua memoria privata, collegate attraverso una rete locale, queste primitive diventano inapplicabili. La conclusione è che i semafori sono troppo a basso livello ed i monitor non sono utilizzabili tranne che in pochi linguaggi di programmazione; inoltre, nessuna delle primitive fornisce lo scambio di informazione tra macchine. È necessario qualcos'altro.

2.3.8 Lo scambio di messaggi

Questo qualcos'altro è lo scambio di messaggi (*message passing*). Questo metodo di comunicazione tra processi usa due primitive, *send* e *receive*, che, come i semafori e a differenza dei monitor, sono chiamate di sistema invece che costrutti del linguaggio. Come tali, possono essere facilmente messe in procedure di libreria, come

```
send(destinazione, &messaggio);
e
receive(sorgente, &messaggio);
```

La prima procedura spedisce un messaggio ad una determinata destinazione e la seconda riceve un messaggio da una determinata sorgente (o da ANY, qualunque sorgente, se al ricevente non interessa una particolare sorgente). Se non è disponibile nessun messaggio, il ricevente potrebbe bloccarsi finché non ne arriva uno; in alternativa, può terminare immediatamente con un codice d'errore.

Problematiche di progetto per sistemi a scambio di messaggi

I sistemi a scambio di messaggi hanno molti problemi avvincenti e problematiche di progetto che non si presentano con i semafori o i monitor, in particolare se i processi che comunicano sono su macchine diverse connesse attraverso una rete; ad esempio, i messaggi possono essere persi dalla rete. Per prevenire la perdita di messaggi, il mittente e il destinatario possono concordare che non appena un messaggio viene ricevuto, il destinatario spedisce indietro uno speciale messaggio di acknowledgement (conferma dell'avvenuta ricezione); se il mittente non ha ricevuto la conferma entro un certo intervallo di tempo, ritrasmette il messaggio.

Consideriamo ora cosa succede se il messaggio stesso è stato ricevuto correttamente, ma la conferma viene persa. Il mittente ritrasmetterà il messaggio, così che il destinatario lo riceverà due volte: è essenziale che il destinatario possa distinguere un nuovo messaggio dalla ritrasmissione di uno vecchio. Di solito questo problema viene risolto mettendo numeri di sequenza consecutivi in ogni messaggio originale; se il destinatario ottiene un messaggio che porta lo stesso numero di sequenza del messaggio precedente, saprà che è un duplicato che può essere ignorato. Comunicare con successo nonostante il passaggio di messaggi sia inaffidabile rappresenta una parte consistente dello studio delle reti di calcolatori. Per maggiori informazioni, si veda Tanenbaum (1996).

I sistemi a scambio di messaggi devono anche trattare il problema dell'assegnazione dei nomi ai processi, in modo che il processo specificato in una chiamata `send` o `receive` non sia ambiguo. Anche l'autenticazione è un problema nei sistemi a scambio di messaggi: come fa il cliente a dire se sta comunicando con il file server reale, e non con un impostore?

All'altro estremo della gamma, ci sono anche problematiche di progetto che sono importanti quando il mittente e il destinatario sono sulla stessa macchina, tra le quali ci sono le prestazioni. La copia di un messaggio da un processo all'altro è sempre più lenta di un'operazione su un semaforo o di un accesso al monitor. È stato fatto molto lavoro per rendere efficiente lo scambio di messaggi. Cheriton (1984), ad esempio, ha suggerito di ridurre la dimensione dei messaggi in modo da contenerli nei registri della macchina, e di usare poi tali registri per lo scambio di messaggi.

Il problema del produttore-consumatore con lo scambio di messaggi

Vediamo ora come può essere risolto il problema del produttore-consumatore con lo scambio di messaggi e senza memoria condivisa: una soluzione viene riportata in Figura 2.29. Supponiamo che tutti i messaggi abbiano la stessa dimensione e che i messaggi spediti ma non ricevuti siano automaticamente bufferizzati dal sistema operativo: si usa un totale di N messaggi, analoghi alle N posizioni di un buffer in memoria condivisa. Il consumatore inizia spedendo N messaggi vuoti al produttore: non appena il produttore ha un elemento da dare al consumatore, prende un messaggio vuoto e ne spedisce indietro uno pieno. In questo modo, il numero totale di messaggi nel sistema rimane costante nel tempo, in modo da memorizzarli in una porzione di memoria predefinita.

Se il produttore lavora più velocemente del consumatore, i messaggi finiranno per essere tutti pieni, in attesa del consumatore; il produttore sarà bloccato, in attesa del ritorno di messaggi vuoti. Se è il consumatore ad essere più veloce, allora accade il viceversa: tutti i messaggi saranno vuoti, in attesa che il produttore li riempia e il consumatore sarà bloccato, in attesa di un messaggio pieno.

Con lo scambio di messaggi sono possibili molte varianti. Per i principianti, vediamo come vengono indirizzati i messaggi; un modo è quello di assegnare ad ogni processo un

indirizzo unico e di indirizzare i messaggi ai processi. Un modo alternativo è quello di inventare una nuova struttura dati, chiamata **mailbox** (letteralmente, cassetta postale), che è un posto per bufferizzare un certo numero di messaggi, di solito specificato quando la mailbox viene creata. Quando si usano le mailbox, i parametri indirizzo nelle chiamate a **send** e **receive** sono **mailbox** e non processi. Quando un processo cerca di spedire un messaggio ad una mailbox piena, viene sospeso finché un messaggio non viene tolto da quella mailbox, facendo spazio per uno nuovo.

Nel problema del produttore-consutatore, sia il produttore che il consumatore dovrebbero creare **mailbox** abbastanza grandi da contenere N messaggi; il produttore dovrebbe spedire messaggi contenenti i dati alla **mailbox** del consumatore e il consumatore dovrebbe spedire messaggi vuoti alla **mailbox** del produttore. Usando le **mailbox**, il meccanismo di bufferizzazione risulta chiaro: la **mailbox** destinazione contiene i messaggi che sono stati spediti al processo destinatario, ma che non sono stati ancora accettati.

L'altro estremo rispetto alle **mailbox** è l'eliminazione della bufferizzazione. Seguendo questo approccio, se la **send** viene chiamata prima della **receive**, il processo mittente viene bloccato finché non viene eseguita la **receive** e, a quel punto, il messaggio può essere copiato direttamente dal mittente al destinatario, senza nessuna bufferizzazione intermedia. In maniera analoga, se la **receive** viene chiamata per prima, il destinatario viene bloccato finché non viene eseguita la **send**. Questa strategia è spesso conosciuta

```
#define N 100                                /* Numero di posizioni nel buffer */

void Produttore(void)
{
    int elemento;
    messaggio m;                                /* Buffer per i messaggi */

    while (TRUE) {
        elemento = produci_elemento();           /* Genera qualcosa da mettere nel buffer */
        receive(Consumatore, &m);                 /* Aspetta che arrivi un messaggio vuoto */
        costruisci_messaggio(&m, elemento);       /* Costruisci un messaggio da spedire */
        send(Consumatore, &m);                    /* Spedisci un elemento al consumatore */
    }
}

void Consumatore(void)
{
    int elemento, i;
    messaggio m;

    for (i = 0; i < N; i++) send(Produttore, &m); /* Spedisci N messaggi vuoti */
    while (TRUE) {
        receive(Produttore, &m);                /* Prendi un messaggio un elemento */
        elemento = estrai_elemento(&m);          /* Estrai un elemento dal messaggio */
        send(Produttore, &m);                    /* Rimanda indietro una risposta vuota */
        consuma_elemento(elemento);             /* Fai qualcosa con l'elemento */
    }
}
```

Figura 2.29 Il problema del produttore-consutatore con N messaggi.

come **rendez-vous** (appuntamento) ed è più facile da implementare rispetto ad uno schema con messaggi bufferizzati, ma è meno flessibile, poiché il mittente e il destinatario sono forzati ad andare in esecuzione a passi sincronizzati.

Lo scambio di messaggi è usato comunemente nei sistemi di programmazione parallela. Un sistema di scambio di messaggi molto conosciuto, ad esempio, è MPI (Message-Passing Interface, interfaccia per lo scambio di messaggi), che è ampiamente usata per i calcoli scientifici. Per maggiori informazioni a questo proposito, si veda, per esempio (Gropp e altri, 1994 e Snir e altri, 1996).

2.3.9 Barriere

Il nostro ultimo meccanismo di sincronizzazione è designato per gruppi di processi, piuttosto che per situazioni del tipo produttore-consumatore a due processi. Alcune applicazioni sono divise in fasi e hanno la regola che nessun processo può proseguire alla prossima fase finché tutti i processi sono pronti per passare alla fase successiva. Questo comportamento può essere ottenuto posizionando una **barriera** alla fine di ogni fase: quando un processo raggiunge la barriera, viene bloccato finché tutti i processi l'hanno raggiunta. L'operazione di una barriera è illustrata in Figura 2.30.

In Figura 2.30(a) vediamo quattro processi che si avvicinano alla barriera, ciò significa che stanno solo facendo calcoli e non hanno ancora raggiunto la fine della fase corrente. Dopo un momento, il primo processo finisce i calcoli richiesti durante la prima fase, quindi esegue la primitiva **barrier**, generalmente chiamando una procedura di libreria, dopodiché il processo viene sospeso. Un attimo dopo, un secondo e poi un terzo processo finiscono la prima fase, ed eseguono anche loro la primitiva **barrier**: questa situazione è illustrata in Figura 2.30(b). Infine, quando l'ultimo processo, C, raggiunge la barriera, tutti i processi vengono rilasciati, come mostrato in Figura 2.30(c).

Come esempio di un problema che richiede le barriere, considerate un problema di rilassamento, tipico in fisica o in ingegneria. Tipicamente, c'è una matrice che contiene qualche valore iniziale; i valori potrebbero rappresentare le temperature in diversi punti su una lastra di metallo, ed il programma potrebbe calcolare quanto tempo impiega una fiamma messa in un angolo a propagarsi attraverso la lastra.

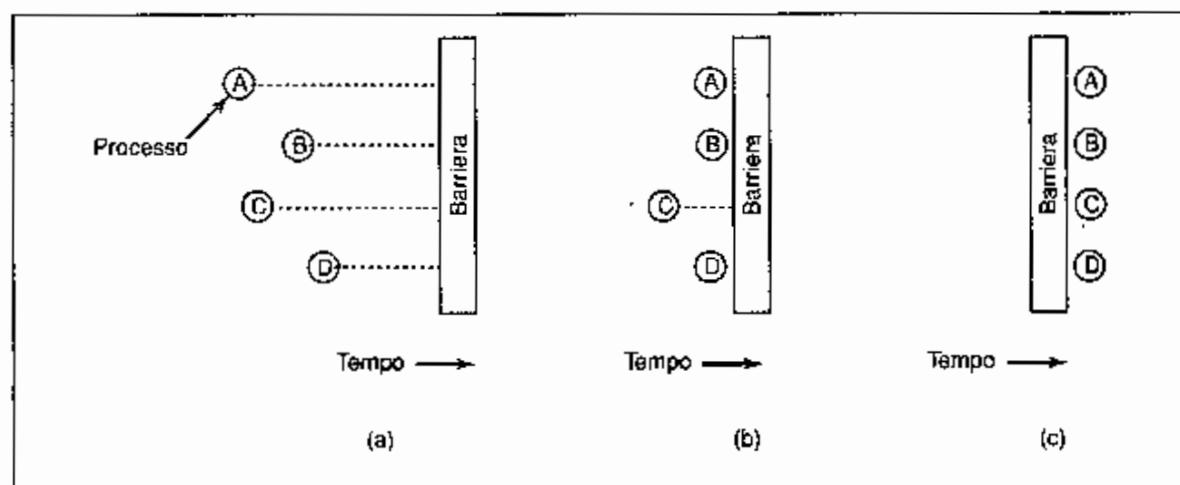


Figura 2.30 Uso di una barriera. [a] I processi si avvicinano alla barriera. [b] Tutti i processi, tranne uno, sono bloccati alla barriera. [c] Quando l'ultimo processo arriva alla barriera, tutti i processi vengono fatti passare.

Cominciando dai valori correnti, si applica una trasformazione alla matrice, ottenendo una seconda versione della matrice, ad esempio applicando le leggi della termodinamica per vedere quali sono tutte le temperature dopo un tempo ΔT . Poi i processi vengono ripetuti più volte, dando le temperature ai punti di rilevazione in funzione del tempo, via via che la lastra si scalda. L'algoritmo produce quindi una serie di matrici nel tempo.

Ora, immaginate che la matrice sia molto ampia (diciamo, un milione per un milione), così che siano necessari processi paralleli (magari su un multiprocessore) per velocizzare il calcolo. Diversi processi lavorano su diverse parti della matrice, calcolando i nuovi elementi della matrice da quelli vecchi, seguendo le leggi della fisica. Comunque, nessun processo può iniziare l'iterazione $n + 1$ finché l'iterazione n è completa, cioè finché tutti i processi hanno terminato il loro lavoro corrente. Il modo per ottenere questo risultato è programmare ogni processo per eseguire un'operazione barrier dopo che ha finito la sua parte dell'iterazione corrente. Quando sono tutte completate, la nuova matrice (i dati in ingresso per la nuova iterazione) sarà finita e tutti i processi saranno rilasciati contemporaneamente per cominciare la prossima iterazione.

2.4 Problemi classici di comunicazione fra processi

La letteratura classica dei sistemi operativi è piena di problemi interessanti che sono stati ampiamente discussi ed analizzati utilizzando una varietà di metodi di sincronizzazione; nelle prossime sezioni esamineremo tre di questi ben noti problemi.

2.4.1 Il problema dei filosofi a cena

Nel 1965, Dijkstra formulò e risolse un problema di sincronizzazione chiamato problema dei filosofi a cena (*dining philosopher problem*). Da allora, chiunque inventi una nuova primitiva di sincronizzazione si è sentito obbligato a dimostrare quanto sia me-

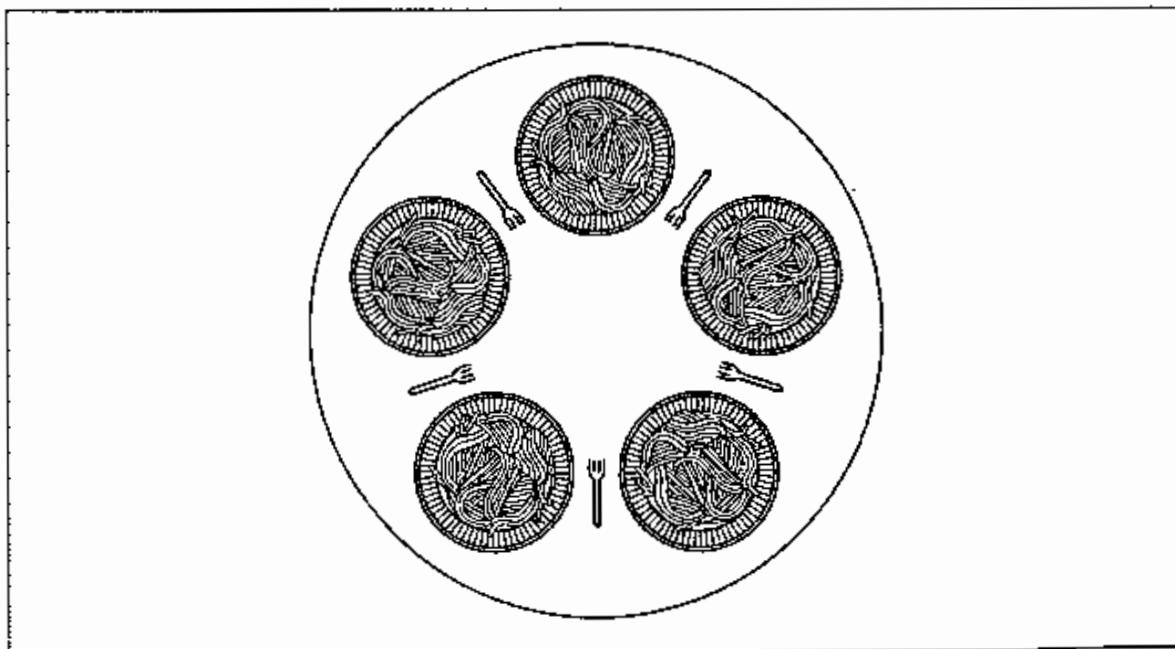


Figura 2.31 L'ora di cena al Dipartimento di Filosofia.

vigliosa la nuova primitiva mostrando quanto elegantemente risolve il problema dei filosofi a cena. Il problema può essere formulato abbastanza semplicemente come segue: cinque filosofi sono seduti attorno ad un tavolo tondo e ciascun filosofo ha un piatto di spaghetti; gli spaghetti sono così scivolosi che per mangiarli ogni filosofo deve avere due forchette; fra ogni coppia di piatti vi è una forchetta. L'aspetto del tavolo è mostrato in Figura 2.31.

La vita dei filosofi alterna periodi in cui essi pensano ad altri in cui mangiano (questa è, in qualche modo, un'astrazione, persino per i filosofi, ma le altre attività qui sono irrilevanti). Quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta che gli sta a sinistra e di quella che gli sta a destra, una alla volta ed in un ordine arbitrario. Qualora riesca a prendere entrambe le forchette, mangia per un po' e, successivamente, depone le forchette e continua a pensare. La domanda chiave è: siete in grado di scrivere un programma per ciascuno dei filosoli che faccia quello che si suppone debba fare e non si fermi mai? (È stato sottolineato come la richiesta di usare due forchette sia in un qualche modo artificiale; forse potremmo spostarci dal cibo italiano a quello cinese, sostituire gli spaghetti con il riso e le forchette con le bacchette).

La Figura 2.32 mostra la soluzione ovvia. La procedura *prendi_forchetta* aspetta fino a che la forchetta specificata è disponibile e ne prende possesso. Sfortunatamente, la soluzione ovvia è sbagliata. Supponete che tutti e cinque i filosofi prendano la loro forchetta sinistra nello stesso istante: nessuno di loro sarà più in grado di prendere la forchetta di destra e, di conseguenza, ci sarà uno stallo.

Potremmo modificare il programma in modo tale che dopo aver preso la forchetta di sinistra, il programma esegua un controllo per vedere se la forchetta di destra è disponibile. Se non lo è, il filosofo rimette a posto quella di sinistra, aspetta per un po' e poi ripete l'intero processo. Anche questa proposta non funziona, sebbene per un motivo diverso. Con un po' di sfortuna, tutti i filosofi potrebbero cominciare l'algoritmo contemporaneamente, prendere le loro forchette di sinistra, accorgersi che la forchetta di destra non è disponibile, rünettere giù la forchetta di sinistra, aspettare, riprendere la forchetta di sinistra simultaneamente e continuare così per sempre. Una situazione come questa, nella quale tutti i programmi continuano ad essere in esecuzione per un tempo indefinito, ma nessuno di essi compie dei veri progressi, è nota come *starvation* (letteralmente, morte per fame). (Viene indicata con lo stesso termine anche in quei casi in cui il problema non capita in un ristorante italiano o cinese.)

```
#define N 5                                /* Numero di filosofi */

void filosofo(int i)                      /* i: quale filosofo, da 0 a 4 */
{
    while (TRUE) {
        pensa();
        prendi_forchetta(i);
        prendi_forchetta((i+1) % N);
        mangia();
        posa_forchetta(i);
        posa_forchetta((i+1) % N);
    }
}
```

Figura 2.32 Una non-soluzione per il problema dei filosofi a cena.

```

#define N      5          /* Numero di filosofi */
#define SINISTRA (i+N-1)%N /* Numero vicino di sinistra di i */
#define DESTRA  (i+1)%N   /* Numero vicino di destra di i */
#define PENSANTE 0         /* Il filosofo sta pensando */
#define AFFAMATO 1         /* Il filosofo cerca di prendere le forchette */
#define MANGIANTE 2        /* Il filosofo sta mangiando */
typedef int semaforo;
int stato[N];
semaforo mutex = 1;
semaforo s[N];
/* I semafori sono un tipo di intero */
/* Vettore per tenere traccia dello stato di ognuno */
/* Mutua esclusione regioni critiche */
/* Un semaforo per filosofo */

void filosofo(int i)           /* i: numero filosofo, da 0 a N - 1 */
{
    while (TRUE) {             /* Ripeti per sempre */
        pensa();
        prendi_forchette(i);
        mangia();
        posa_forchette(i);
    }
}

void prendi_forchette(int i)    /* i: numero filosofo, da 0 a N - 1 */
{
    down(&mutex);
    stato[i] = AFFAMATO;
    test(i);
    up(&mutex);
    down(&s[i]);
    /* Si blocca se forchette non ottenute */
}

void posa_forchette(i)         /* i: numero filosofo , da 0 a N - 1 */
{
    down(&mutex);
    stato[i] = PENSANTE;
    test(SINISTRA);
    test(DESTRA);
    up(&mutex);
    /* Esce dalla regione critica */
}

void test(i)                  /* i: numero filosofo , da 0 a N - 1 */
{
    if (stato[i] == AFFAMATO && stato[SINISTRA] != MANGIANTE && stato[DESTRA] != MANGIANTE)
    {
        stato[i] = MANGIANTE;
        up(&s[i]);
    }
}

```

Figura 2.33 Una soluzione al problema dei filosofi a cena.

Adesso voi potreste pensare: "E se i filosofi aspettassero un tempo casuale anziché un tempo fisso fra un tentativo e l'altro di accedere alle forchette, la probabilità che tutto possa andare avanti a bloccarsi anche per una sola ora sarebbe molto bassa." Questa osservazione è vera e in quasi tutte le applicazioni riprovare più tardi non è un problema. Ad esempio, nella popolare rete locale Ethernet, se due calcolatori spediscono un pacchetto contemporaneamente, ognuno aspetta un tempo casuale e riprova; in pratica, questa soluzione lavora bene. Comunque, in alcune applicazioni si vuole una soluzione che funzioni sempre e che non possa fallire a causa di una serie sfortunata di numeri casuali. Si pensi al controllo di sicurezza di una centrale nucleare.

Un miglioramento rispetto alla Figura 2.32 che non presenta né stallo né attesa infinita, è quello di proteggere le cinque istruzioni che seguono la chiamata alla *pensa* tramite un semaforo binario. Prima di cominciare a prendere possesso delle forchette, un filosofo dovrebbe fare una *down* su *mutex*. Dopo aver deposto le forchette, dovrebbe fare una *up* su *mutex*. Da un punto di vista teorico, questa soluzione è adeguata, ma da un punto di vista pratico, presenta un problema di prestazioni: un solo filosofo alla volta è in grado di mangiare. Con cinque forchette a disposizione, dovremmo invece essere in grado di far mangiare due filosofi alla volta.

La soluzione presentata nella Figura 2.33 è libera dallo stallo e permette il massimo grado di parallelismo per un numero arbitrario di filosofi. Usa un vettore, *affamato*, per tener traccia se un filosofo sta mangiando, pensando o se sia affamato (cioè se sta cercando di ottenere le forchette). Un filosofo si può portare nello stato in cui mangia solo se nessuno dei suoi vicini sta mangiando. I vicini del filosofo i sono definiti tramite le macro *SINISTRA* e *DESTRA*; in altre parole, se *i* vale 2, *SINISTRA* vale 1 e *DESTRA* vale 3.

Il programma usa un vettore di semafori, uno per filosofo, così che i filosofi affamati possano bloccarsi se le forchette di cui hanno bisogno risultano occupate. Si noti che ogni processo esegue come programma principale la stessa procedura *filosofo*, ma che le altre procedure, *prendi_forchette*, *posa_forchette* e *test* sono procedure ordinarie e non processi separati.

2.4.2 Il problema dei lettori e scrittori

Il problema dei filosofi a cena è utile per modellare processi che competono per l'accesso esclusivo ad un numero limitato di risorse, come i dispositivi di ingresso/uscita. Un altro problema noto è quello dei lettori e scrittori (Courtois e altri, 1971), che modella l'accesso ad una base di dati. Immaginate, ad esempio, un sistema di prenotazione di una linea aerea, con tanti processi in competizione che desiderano leggervi e scrivervi, ma se un processo sta aggiornando (scrivendo) la base di dati, nessun altro processo può avere accesso alla base di dati, nemmeno i lettori. La domanda è: come programmare i lettori e gli scrittori? Una soluzione è mostrata in Figura 2.34.

In questa soluzione, il primo lettore che ottiene l'accesso alla base di dati esegue una *down* sul semaforo *db*. I lettori successivi incrementano semplicemente un contatore, *rc*; via via che i lettori escono, decrementano il contatore e l'ultimo esegue una *up* sul semaforo, permettendo ad uno scrittore bloccato, se esiste, di accedere alla base di dati.

La soluzione qui presentata contiene implicitamente una sottile decisione che vale la pena commentare: supponete che, mentre un lettore sta usando la base di dati, si presenti un altro lettore: dal momento che avere due lettori contemporaneamente non crea problemi, il secondo lettore è ammesso, ed anche un terzo ulteriore lettore può essere ammesso, se si presenta.

Ora, supponete che si presenti uno scrittore: esso non può accedere alla base di dati,

dal momento che devono avere accesso esclusivo, così che lo scrittore viene sospeso. Più tardi, altri lettori compaiono, e finché l'ultimo lettore è ancora attivo, ulteriori lettori sono consentiti. Come conseguenza di questa strategia, finché c'è un regolare arrivo di lettori, questi avranno immediatamente accesso alla base di dati, mentre lo scrittore rimarrà sospeso finché non ci sono più lettori. Se arriva un nuovo lettore, supponiamo, ogni 2 secondi, e ogni lettore impiega 5 secondi a fare il suo lavoro, lo scrittore non avrà mai accesso alla base di dati.

Per evitare questa situazione, il programma dovrebbe essere leggermente diverso: quando un lettore arriva e uno scrittore sta aspettando, il lettore viene sospeso dietro lo scrittore, invece di essere ammesso immediatamente. In questo modo, uno scrittore deve aspettare che i lettori che erano attivi quando è arrivato finiscano, ma non deve aspettare i lettori che si presentano dopo di lui. Lo svantaggio di questa soluzione è che realizza meno concorrenza e, quindi, peggiori prestazioni. Courtois e altri presentano una soluzione che dà priorità agli scrittori. Per i dettagli, rimandiamo all'articolo.

```

typedef int semaforo;           /* Indovina? */
semaforo mutex = 1;             /* Controlla gli accessi a 'rc' */
semaforo db = 1;                /* Controlla accessi alla base dati */
int rc = 0;                     /* Numero processi che leggono o vogliono leggere */

void lettore(void)
{
    while (TRUE) {           /* Ripeti all'infinito */
        down(&mutex);         /* Ottieni accesso esclusivo a 'rc' */
        rc = rc + 1;           /* Un lettore in più ora */
        if (rc == 1) down(&db); /* Se questo è il primo lettore ... */
        up(&mutex);            /* Rilascia accesso esclusivo a 'rc' */
        leggi_base_dati();     /* Accedi ai dati */
        down(&mutex);           /* Ottieni accesso esclusivo a 'rc' */
        rc = rc - 1;           /* Un lettore in meno, ora */
        if (rc == 0) up(&db);   /* Se questo è l'ultimo lettore ... */
        up(&mutex);            /* Rilascia accesso esclusivo ad 'rc' */
        leggi_dati_utente();   /* Sezione non critica */
    }
}

void scrittore(void)
{
    while (TRUE) {           /* Ripeti all'infinito */
        pensa_ai_dati();      /* Sezione non critica */
        down(&db);             /* Ottieni l'accesso esclusivo */
        scrivi_base_dati();    /* Aggiorna i dati */
        up(&db);                /* Rilascia l'accesso esclusivo */
    }
}

```

Figura 2.34 Una soluzione al problema dei lettori e scrittori.

2.4.3 Il problema del barbiere che dorme

Un altro tipico problema relativo alla comunicazione fra processi è ambientato in un negozio di barbiere. Nel negozio c'è un unico barbiere, una sedia da lavoro e n sedie per far sedere i clienti in attesa, quando ce ne sono; se non ci sono clienti, il barbiere si siede sulla sedia da lavoro e ci si addormenta, come in Figura 2.35. Quando arriva un cliente, deve svegliare il barbiere; se poi arrivano altri clienti mentre il barbiere taglia i capelli a un cliente, essi attendono seduti (se ci sono sedie libere) oppure lasciano il negozio (se tutte le sedie sono occupate). Il problema è quello di programmare il barbiere ed i clienti senza arrivare ad avere corse critiche; è simile alle varie situazioni in cui si ha una coda, come un helpdesk con più persone con un sistema di attesa a chiamata computerizzata per mantenere un numero limitato di chiamate in arrivo.

Una soluzione usa tre semafori: *clienti*, che conta il numero di clienti in attesa (escluso il cliente sulla sedia da lavoro, che non sta aspettando), *barbieri*, il numero di barbiere (0 o 1) inattivi, in attesa di clienti e *mutex*, che è usato per la mutua esclusione. Abbiamo anche bisogno di una variabile, *in_attesa*, che conta anch'essa il numero di clienti in attesa, e che è essenzialmente una copia di *clienti* (il motivo per cui si introduce *in_attesa* è che non c'è nessun modo per leggere il valore corrente di un semaforo e, in questa soluzione, un cliente che entra nel negozio deve contare il numero di clienti in attesa; se è infe-

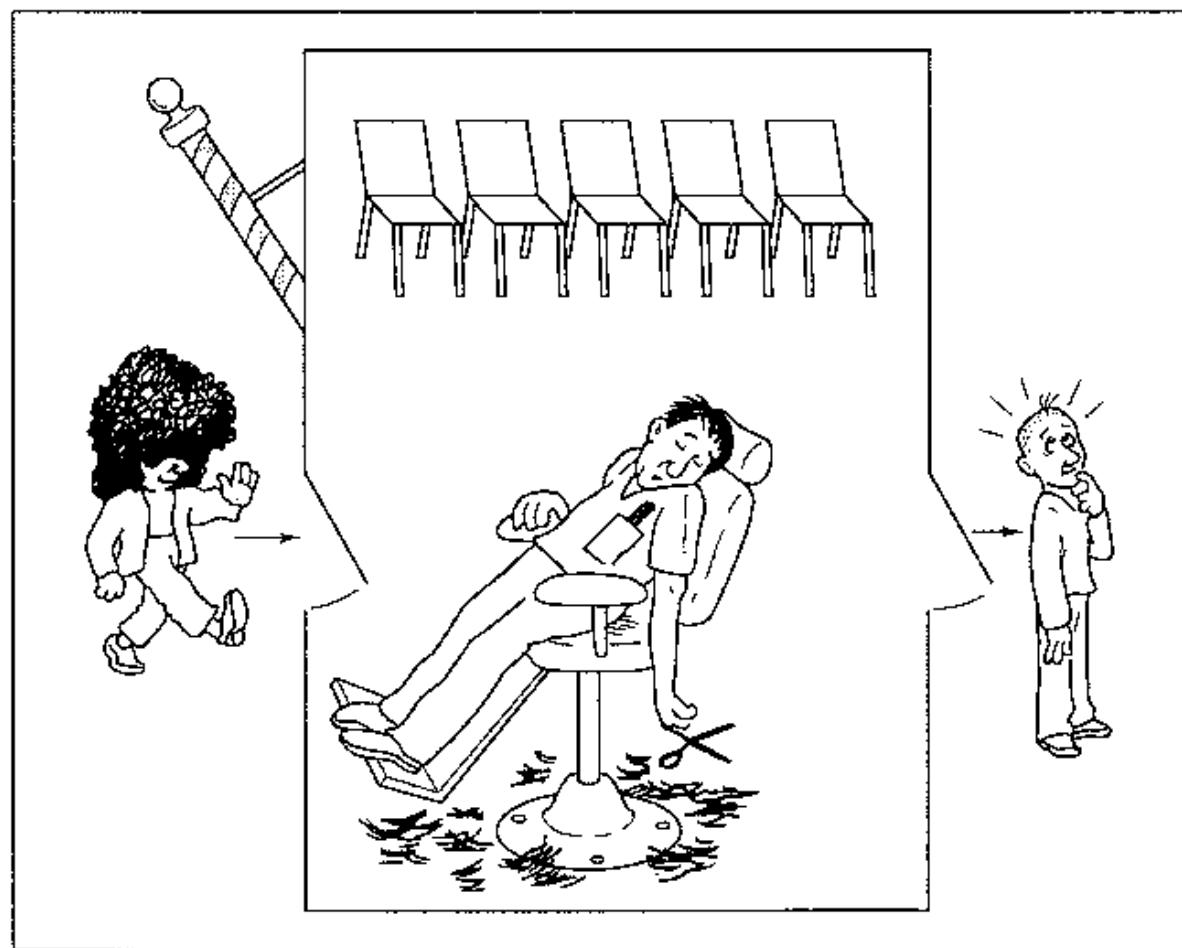


Figura 2.35 Il barbiere addormentato.

riore al numero di sedie, il cliente rimane, altrimenti se ne va).

La nostra soluzione è mostrata in Figura 2.36. Quando la mattina il barbiere apre il negozio, esegue la procedura *barbiere*, che lo blocca sul semaforo *clienti*, perché è inizialmente 0. Egli va quindi a dormire come mostrato in Figura 2.35, e rimane addormentato finché non arriva il primo cliente.

Quando arriva il primo cliente, questi esegue *cliente*, iniziando con l'acquisizione di *mutex* per entrare nella regione critica. Se subito dopo arriva un altro cliente, questo non sarà in grado di fare niente finché il primo non avrà rilasciato *mutex*. Il cliente controlla poi se il numero dei clienti in attesa è minore del numero di sedie; se non lo è, rilascia *mutex* e se ne va senza il taglio dei capelli.

Se c'è una sedia disponibile, il cliente incrementa la variabile intera *in_attesa*; poi esegue una *up* sul semaforo *clienti*, risvegliando il barbiere; a questo punto, il cliente ed il barbiere sono entrambi svegli. Quando il cliente rilascia *mutex*, il barbiere lo acquisisce, fa un po' di attività di servizio, e inizia a tagliare i capelli.

```
#define SEDIE 5           /* Numero sedie per clienti in attesa */

typedef int semaforo;    /* Indovina? */

semaforo clienti = 0;    /* Numero clienti in attesa */
semaforo barbieri = 0;   /* Numero barbieri in attesa di clienti */
semaforo mutex = 1;      /* Per la mutua esclusione */
int in_attesa = 0;        /* Clienti in attesa (non durante il taglio) */

void barbiere(void)
{
    while (TRUE) {
        down(&clienti);          /* Sospenditi se numero clienti zero */
        down(&mutex);            /* Acquisisci accesso a in_attesa */
        in_attesa = in_attesa - 1; /* Decrementa contatore clienti in attesa */
        up(&barbieri);           /* Un barbiere è pronto per il taglio */
        up(&mutex);              /* Rilascia in_attesa */
        taglia_capelli( );       /* Taglia capelli (fuori sezione critica) */
    }
}

void cliente(void)
{
    down(&mutex);             /* Entra nella regione critica */
    if (in_attesa < CRAIES) {  /* Se non ci sono sedie libere, vattene */
        in_attesa = in_attesa + 1; /* Incrementa numero di clienti in attesa */
        up(&clienti);           /* Sveglia un barbiere, se necessario */
        up(&mutex);              /* Rilascia l'accesso a in_attesa */
        down(&barbieri);         /* Sospenditi se numero barbieri liberi zero */
        vai_al_taglio( );        /* Siediti per essere servito */
    } else {
        up(&mutex);              /* Il negozio è pieno; non attendere */
    }
}
```

Figura 2.36 Una soluzione al problema del barbiere che dorme.

Quando il taglio dei capelli è concluso, il cliente termina la procedura e lascia il negozio. A differenza dei nostri esempi precedenti, il cliente non esegue alcun ciclo, poiché ciascun cliente si fa tagliare i capelli una volta sola. Invece, il barbiere esegue un ciclo e cerca di servire il prossimo cliente: se ce n'è uno presente, fa un altro taglio di capelli, altrimenti il barbiere va a dormire.

Detto per inciso, vale la pena di evidenziare che, sebbene il problema dei lettori e scrittori e quello del barbiere che dorme non richiedano trasferimento di dati, appartengono entrambi all'area dei problemi di comunicazione fra processi, perché richiedono sincronizzazione fra molti processi.

2.5 Schedulazione dei processi

Quando il calcolatore è multiprogrammato, ha spesso diversi processi che competono per l'uso della CPU contemporaneamente: tale situazione si verifica ogniqualvolta due o più processi sono contemporaneamente nello stato di pronto. Se è disponibile solo una CPU, bisogna scegliere il prossimo processo da eseguire; la parte del sistema operativo che si occupa di questa decisione viene chiamata **schedulatore** (*scheduler*) e l'algoritmo che usa è chiamato **algoritmo di schedulazione** (*scheduling algorithm*). Questo argomento è l'oggetto delle sezioni seguenti.

Molte delle questioni relative alla schedulazione dei processi si applicano anche alla schedulazione dei thread, sebbene ci siano delle differenze. All'inizio ci focalizzeremo sulla schedulazione dei processi, più tardi daremo esplicitamente un'occhiata alla schedulazione dei thread.

2.5.1 Introduzione alla schedulazione

Ai vecchi tempi dei sistemi batch, con dati in ingresso in forma di immagini delle schede perforate su nastro magnetico, l'algoritmo di schedulazione era semplice: bastava eseguire il prossimo job sul nastro. Con i sistemi timesharing, l'algoritmo di schedulazione divenne più complesso, perché c'erano, in generale, più utenti in attesa di un servizio. Alcuni mainframe combinano ancora servizi batch e timesharing, richiedendo allo schedulatore di decidere se il prossimo dovrebbe essere un job batch oppure un utente interattivo al terminale. (Detto per inciso, un job batch può essere una richiesta di esecuzione di molti programmi in sequenza, ma, in questa sezione, supporremo che sia solo una richiesta di esecuzione di un solo programma.) Poiché il tempo di CPU è una risorsa scarsa su queste macchine, un buon schedulatore può fare la differenza in termini di prestazioni percepibili e soddisfazione dell'utente. Di conseguenza, è stato speso molto lavoro per escogitare algoritmi di schedulazione efficienti ed ingegnosi.

Con l'avvento dei personal computer, la situazione è cambiata in due modi. Primo, la maggior parte del tempo c'è solo un processo attivo: se un utente sta modificando un documento con un elaboratore di testi, è improbabile che stia contemporaneamente compilando un programma in background. Quando un utente batte un comando dell'elaboratore di testi, lo schedulatore non ha molto lavoro da fare per riuscire a capire quale processo deve eseguire, perché l'elaboratore di testi è il solo candidato.

Secondo, i calcolatori sono diventati così veloci nel corso degli anni che raramente la CPU è ancora una risorsa scarsa. La maggior parte dei programmi per personal computer sono limitati dalla velocità a cui l'utente può presentare dati in ingresso (scrivendo o cliccando), non dalla velocità con cui la CPU può elaborarli. Persino le compilazioni, una del-

le maggiori perdite di cicli di CPU nel passato, oggi richiedono alcuni secondi al massimo. Anche quando due programmi vengono realmente eseguiti allo stesso istante, come un elaboratore di testi e un foglio elettronico, poco importa quale dei due va prima, poiché l'utente sta aspettando che entrambi finiscano, probabilmente. Di conseguenza la schedulazione non ha molta importanza su un PC semplice. [Certamente esistono applicazioni che praticamente si mangiano viva la CPU: riprodurre un'ora di video ad alta qualità può richiedere un'elaborazione a livello industriale delle immagini su ognuno dei 108 000 fotogrammi in NTSC (90000 in PAL), ma queste applicazioni sono l'eccezione più che la regola.]

Quando si passa alle postazioni di lavoro e ai server di fascia alta collegati in rete, la situazione cambia: in questo caso molti processi spesso si contendono la CPU, quindi la schedulazione torna ad avere importanza. Per esempio, quando la CPU deve scegliere tra eseguire un processo che aggiorna lo schermo dopo che un utente ha chiuso una finestra, ed eseguire un processo che spedisce le e-mail in coda, fa un'enorme differenza nella risposta percepita. Se la chiusura di una finestra dovesse impiegare 2 secondi mentre l'e-mail viene spedita, l'utente probabilmente vedrebbe il sistema come molto pigro, mentre avere un'e-mail ritardata di 2 secondi non verrebbe notato nemmeno. In questo caso la schedulazione dei processi ha molta importanza.

Oltre a scegliere il processo giusto da eseguire, lo scheduler deve preoccuparsi di fare un uso efficiente della CPU, perché il cambio di processo è costoso. Tanto per cominciare, deve cambiare dalla modalità utente alla modalità kernel; quindi deve salvare lo stato del processo corrente, memorizzando i suoi registri nella tabella dei processi in modo che possano essere ricaricati in seguito. In molti sistemi, deve anche essere salvata la mappa di memoria (per esempio i bit di riferimento alla memoria nella tabella delle pagine). Quindi deve scegliere un nuovo processo dall'algoritmo di schedulazione corrente; dopo questo, deve ricaricare la MMU con la mappa di memoria del nuovo processo, ed infine, deve far ripartire il nuovo processo. In aggiunta a tutto questo, il cambio di processo spesso invalida l'intera cache di memoria, forzando a ricaricarla dinamicamente dalla memoria due volte (entrando nel kernel e uscendone). Tutto sommato, fare troppi cambi di processo per secondo può consumare una considerevole quantità di tempo di CPU, quindi è consigliabile fare attenzione.

Comportamento dei processi

Quasi tutti i processi alternano raffiche di calcoli con richieste di operazioni di ingresso/uscita (del disco), come mostrato in Figura 2.37. Di solito la CPU lavora per un po' senza fermarsi, quindi esegue una chiamata di sistema per leggere da un file o per scrivere su un file. Quando la chiamata di sistema è completata, la CPU esegue di nuovo calcoli, fino a che ha bisogno di ulteriori dati o di scrivere altri e così via. Si noti che qualche attività di ingresso/uscita conta come calcolo: per esempio, quando la CPU copia i bit sulla RAM del video per aggiornare lo schermo, sta facendo calcoli, non ingresso/uscita, perché è la CPU ad essere in uso. In questo senso l'ingresso/uscita si ha quando un processo entra nello stato bloccato in attesa che un dispositivo esterno completi il suo lavoro.

La cosa importante da notare a proposito della Figura 2.37 è che qualche processo, come quello in Figura 2.37(a), spende la maggior parte del suo tempo in calcoli, mentre altri, come quello in Figura 2.37(b), la spendono in attesa di ingresso/uscita. I primi sono chiamati processi **compute bound** o **CPU bound** (orientati ai calcoli), mentre i secondi sono chiamati **I/O bound** (orientati alle operazioni di ingresso/uscita). I processi orientati ai calcoli hanno tipicamente lunghi tratti di uso di CPU e quindi attese di ingres-

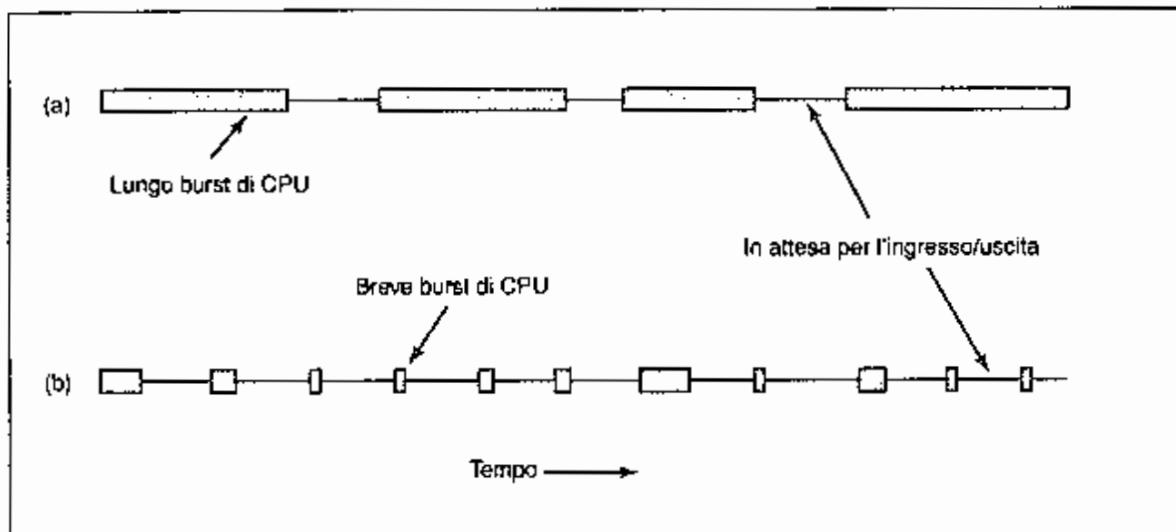


Figura 2.37 Raffiche di utilizzo di CPU alternate a periodi di attesa di ingresso/uscita. (a) Un processo CPU bound. (b) Un processo I/O bound.

so/uscita poco frequenti, mentre i processi orientati all'ingresso/uscita hanno brevi tratti di uso di CPU e frequenti attese di ingresso/uscita. Si noti che il fattore chiave è la lunghezza del tratto di uso di CPU, non la lunghezza dell'attesa di ingresso/uscita. I processi orientati all'ingresso/uscita sono chiamati così perché non fanno molti calcoli tra le richieste di ingresso/uscita. Ci vuole sempre lo stesso tempo per leggere un blocco dal disco, non importa quanto tempo o quanto poco tempo serve per elaborare i dati dopo che sono arrivati. Bisogna notare che più la CPU si velocizza, più i processi tendono ad essere orientati all'ingresso/uscita.

Questo effetto si ha perché le CPU migliorano più velocemente che i dischi; di conseguenza, è probabile che la schedulazione dei processi orientati all'ingresso/uscita diventi un argomento più importante in futuro. In questo caso l'idea base è che, se un processo orientato all'ingresso/uscita vuole essere eseguito, dovrebbe avere una possibilità entro breve tempo, in modo che possa impartire la sua richiesta al disco tenendo il disco occupato.

Quando schedulare

Una questione chiave collegata alla schedulazione è quando prendere le decisioni di schedulazione; esistono un sacco di situazioni in cui è necessario schedulare. Primo, quando viene creato un nuovo processo, si deve decidere se eseguire il processo genitore o il processo figlio. Dal momento che entrambi i processi sono nello stato di pronto, è una decisione di schedulazione normale, e può essere scelto l'uno o l'altro, cioè lo scheduler può legittimamente scegliere di eseguire o il genitore o il figlio.

Secondo, una decisione di schedulazione deve essere presa quando un processo termina: il processo non può più essere eseguito (dal momento che non esiste più), così bisogna scegliere qualche altro processo dall'insieme dei processi che si trovano in stato di pronto. Se nessun processo è in questo stato, viene normalmente eseguito il processo inattivo fornito dal sistema.

Terzo, quando un processo si blocca su un'operazione di ingresso/uscita, su un semaforo, o per qualche altra ragione, si deve scegliere un altro processo per l'esecuzione.

ne. Qualche volta il motivo per cui il processo si è bloccato può essere significativo per la scelta da fare: per esempio, se *A* è un processo importante e sta aspettando che *B* esca dalla sua regione critica, eseguire *B* come prossimo processo gli permette di uscire dalla sua regione critica, e quindi permette ad *A* di continuare. Il guaio, comunque, è che lo scheduler, generalmente, non ha le informazioni necessarie per prendere in considerazione questa dipendenza.

Quarto, deve essere presa una decisione di schedulazione quando si verifica un'interruzione di ingresso/uscita: se essa proviene da un dispositivo di ingresso/uscita che ha ora completato il suo lavoro, il processo che era stato bloccato in attesa di quest'operazione è ora pronto per essere eseguito. È compito dello scheduler decidere se debba essere eseguito il processo che è appena passato nello stato pronto, il processo che era in esecuzione al momento in cui si è verificata l'interruzione, o un terzo processo.

Se il clock hardware fornisce interruzioni periodiche a 50 Hz, 60 Hz o a una qualche altra frequenza, si deve prendere una decisione di schedulazione ad ogni interruzione di clock, o dopo k interruzioni. Gli algoritmi di schedulazione si possono dividere in due categorie, tenendo in considerazione come questi trattano le interruzioni di clock. Un algoritmo di schedulazione nonpreemptive (senza prerilascio) sceglie un processo da eseguire e quindi semplicemente lo lascia in esecuzione finché questo non si blocca, su un'operazione di ingresso/uscita o in attesa di un altro processo, oppure finché questo non rilascia volontariamente la CPU. Persino se questo viene eseguito per ore, non verrà forzatamente sospeso; in realtà, non si prende nessuna decisione di schedulazione durante le interruzioni di clock, e dopo aver completata la procedura di interruzione di clock, viene sempre ripreso il processo che era in esecuzione prima dell'interruzione.

Diversamente, un algoritmo di schedulazione preemptive (con prerilascio), sceglie un processo e lo lascia in esecuzione al massimo per una quantità prefissata di tempo. Se questo è ancora in esecuzione allo scadere dell'intervallo di tempo, viene sospeso e lo scheduler sceglie un altro processo da eseguire (se ce n'è uno disponibile). Effettuare la schedulazione con prerilascio richiede un'interruzione di clock alla fine dell'intervallo di tempo, per restituire il controllo della CPU allo scheduler. Se non è disponibile nessun clock, la sola possibilità di scelta è la schedulazione senza prerilascio.

Tipi di algoritmi di schedulazione

Non è sorprendente che diversi ambienti richiedano diversi algoritmi di schedulazione; tale situazione si presenta perché diverse aree applicative (e diversi tipi di sistema operativo) hanno scopi diversi; in altre parole, quello che lo scheduler deve ottimizzare non è la stessa cosa in tutti i sistemi. Tre ambienti che vale la pena di distinguere sono:

1. Batch.
2. Interattivo.
3. Real time.

Nei sistemi batch, non ci sono utenti impazienti in attesa ai loro terminali di una risposta veloce. Di conseguenza, sono spesso accettabili sia algoritmi senza prerilascio sia algoritmi con prerilascio con un periodo lungo per ogni processo. Questo approccio riduce gli scambi fra i processi, migliorando quindi le prestazioni.

In un ambiente con utenti interattivi, il prerilascio (preemption) è essenziale per evitare che un processo si impossessi della CPU, impedendo agli altri processi di ottenere servizi. Anche se nessun processo vuole intenzionalmente essere eseguito per sempre,

a causa di un errore di programma un processo potrebbe escludere tutti gli altri per un tempo indeterminato. È necessario quindi il prerilascio per impedire questo comportamento.

In sistemi con vincoli real time, il prerilascio, abbastanza stranamente, qualche volta, non è necessario, perché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e normalmente fanno il loro lavoro e si bloccano in fretta. La differenza con i sistemi interattivi è che i sistemi real time eseguono solo programmi che agevolano le altre applicazioni, mentre i sistemi interattivi sono d'uso generale e possono eseguire programmi arbitrari che non collaborano o sono persino dolosi.

Obiettivi degli algoritmi di schedulazione

Per progettare un algoritmo di schedulazione, è necessario avere qualche idea di cosa dovrebbe fare un buon algoritmo. Qualche obiettivo dipende dall'ambiente (batch, interattivo o real time), ma ce ne sono alcuni che sono opportuni in tutti i casi. Alcuni obiettivi sono elencati in Figura 2.38: li discuteremo a turno più sotto.

In tutte le circostanze, l'equità è importante. Processi simili dovrebbero ottenere servizi paragonabili, quindi non è corretto assegnare ad un processo più tempo di CPU rispetto a un processo equivalente. Ovviamemente, categorie diverse di processi possono essere trattate in modo molto diverso; pensate al controllo di sicurezza ed a calcolare il libro paga al centro di calcolo di un reattore nucleare.

Collegata all'equità è l'applicazione delle politiche di sistema. Se la politica locale è che i processi di controllo della sicurezza possano essere eseguiti quando vogliono, persino se questo significa che il libro paga è in ritardo di 30 secondi, lo scheduler deve essere sicuro che questa politica venga rispettata.

Un altro obiettivo generico è quello di tenere, quando possibile, tutte le parti dei

Tutti i sistemi
Equità: dare a ogni processo una porzione equa di CPU Applicazione delle politiche di sistema: vedere che la politica stabilita venga messa in atto Bilanciamento: tenere occupate tutte le parti del sistema
Sistemi batch
Throughput: massimizzare i job all'ora Tempo di turnaround: minimizzare il tempo fra la richiesta e la conclusione Uso della CPU: tenere occupata la CPU tutto il tempo
Sistemi interattivi
Tempo di risposta: rispondere velocemente alle richieste Proportionalità: andare incontro alle aspettative degli utenti
Sistemi real-time
Rispettare le scadenze: evitare la perdita di dati Prevedibilità: evitare la degradazione qualitativa nei sistemi multimediali

Figura 2.38 Alcuni obiettivi degli algoritmi di schedulazione in condizioni diverse.

sistema occupate, perché se la CPU e tutti i dispositivi di ingresso/uscita possono essere tenuti in esecuzione tutto il tempo, verrà fatto più lavoro al secondo che non se alcune componenti riunangono inattive (idle). In un sistema batch, ad esempio, lo scheduler ha il controllo di quali job vengono portati in memoria per essere eseguiti. Avere qualche processo CPU bound (che usa pesantemente la CPU) e qualche processo I/O bound (che prevalentemente interagisce con i dispositivi di ingresso/uscita) contemporaneamente in memoria è un'idea migliore che prima caricare ed eseguire tutti i job CPU bound e poi, quando questi hanno finito, caricare ed eseguire tutti i job I/O bound. Se viene utilizzata la seconda strategia, quando sono in esecuzione processi CPU bound, si contenderanno la CPU e il disco rimarrà inattivo; dopo, quando entrano in gioco i job I/O bound, si contenderanno il disco e la CPU sarà inattiva. Meglio tenere in esecuzione l'intero sistema nello stesso momento mescolando per bene i processi.

I gestori dei grossi centri di calcolo che eseguono molti job batch di solito considerano tre metriche per vedere quanto lavorano bene i loro sistemi: throughput (produttività), il tempo di turnaround e l'uso della CPU. Il throughput è il numero dei job per ora che il sistema completa: tutto considerato, terminare 50 job in un'ora è meglio che terminarne 40. Il tempo di turnaround è il tempo medio statistico dal momento che viene richiesto un job batch fino al momento che il job viene completato e misura quanto tempo, in media, l'utente deve aspettare per ottenere i dati in uscita. Qui la regola è: piccolo è bellissimo.

Un algoritmo di schedulazione che massimizza il throughput non necessariamente minimizza il tempo di turnaround. Ad esempio, dato un mix di job brevi e lunghi, uno scheduler che esegue sempre job brevi e non esegue mai quelli lunghi potrebbe ottenere un throughput eccellente (molti job brevi all'ora), ma a spese di un tempo di turnaround tremendo per i job lunghi. Se i job brevi arrivano a frequenza regolare, i job lunghi potrebbero non essere mai eseguiti, rendendo infinito il tempo di turnaround medio, ottenendo invece un alto throughput.

L'uso della CPU è un problema anche con i sistemi batch, perché sui grossi mainframe dove vengono eseguiti sistemi batch, la CPU rappresenta ancora la spesa principale, così gli amministratori dei centri di calcolo si sentono in colpa quando essa non è sempre attiva. In realtà, questo non è un buon parametro, quello che importa davvero è quanti job all'ora escono dal sistema (throughput) e quanto tempo richiede l'uscita di un job (tempo di turnaround). Considerare l'uso della CPU come parametro è come valutare le auto basandosi su quanti giri all'ora fa il motore.

Per i sistemi interattivi, in particolar modo sistemi timesharing e server, si applicano diversi obiettivi; il più importante è minimizzare il tempo di risposta, che è il tempo fra l'invio di un comando e l'arrivo del risultato. Su un personal computer, dove è in esecuzione un processo in background, (ad esempio, leggere e memorizzare e-mail da una rete), una richiesta utente di iniziare un programma o aprire un file dovrebbe avere precedenza sul lavoro in background; se tutte le richieste interattive sono eseguite per prime, verrà percepito un buon servizio.

Un problema in qualche modo correlato è quello che potrebbe essere chiamato proporzionalità. Gli utenti hanno un'idea insita (ma spesso sbagliata) del tempo che le attività dovrebbero richiedere; quando una richiesta che è percepita come complessa richiede molto tempo, gli utenti lo accettano, ma quando una richiesta che viene considerata semplice richiede molto tempo, gli utenti si irritano. Ad esempio, se cliccare su un'icona che chiama un provider Internet utilizzando un modem analogico richiede 45 secondi per stabilire la connessione, l'utente, probabilmente, lo accetterà come un fatto della vita. D'altro canto, se cliccare su un'icona che interrompe la connessione richiede 45 secon-

di, l'utente probabilmente entro 30 secondi starà imprecando e entro 45 secondi avrà la schiuma alla bocca. Questo comportamento è dovuto alla percezione dell'utente comune che fare una telefonata e ottenere una connessione si suppone che richieda di più che semplicemente riattaccare. In alcuni casi (come questo), lo scheduler non può fare niente per il tempo di risposta, ma in altri casi può, in particolare quando il ritardo è dovuto a una scelta scadente per l'ordine dei processi.

I sistemi real time hanno caratteristiche diverse dai sistemi interattivi, quindi hanno diversi obiettivi di schedulazione, in quanto sono caratterizzati dall'avere scadenze che devono, o almeno dovrebbero, essere rispettate. Ad esempio, se un calcolatore sta controllando un dispositivo che produce dati a una frequenza regolare, errori nell'esecuzione puntuale del processo di raccolta dei dati possono provocare una perdita di dati. Così, la principale necessità di un sistema real time è rispettare tutte (o la maggior parte) le scadenze.

In alcuni sistemi real time, in particolare quelli che coinvolgono multimedia, è importante la prevedibilità. Non rispettare una scadenza ogni tanto non è grave, ma se il processo audio viene eseguito in modo troppo irregolare, la qualità del suono deteriorerà rapidamente. Anche il video è un problema, ma l'udito è molto più sensibile al jitter (tremolio) che non l'occhio. Per evitare questo problema, la schedulazione dei processi deve essere molto prevedibile e regolare. In questo capitolo studieremo gli algoritmi di schedulazione batch e interattivi, ma rimanderemo la maggior parte del nostro studio della schedulazione real time finché non incontreremo i sistemi operativi multimediali, nel Capitolo 7.

2.5.2 Schedulazione nei sistemi batch

È ora tempo di passare dai problemi generici di schedulazione agli algoritmi specifici di schedulazione; in questa sezione, daremo un'occhiata agli algoritmi utilizzati nei sistemi batch, mentre nelle seguenti esamineremo i sistemi interattivi e real time. Vale la pena di sottolineare che alcuni algoritmi vengono utilizzati sia nei sistemi batch sia interattivi, e li studieremo più avanti; qui ci focalizzeremo sugli algoritmi che sono adatti solo ai sistemi batch.

First-Come First-Served

Probabilmente il più semplice di tutti gli algoritmi di schedulazione è l'algoritmo senza prerilascio **primo arrivato, primo servito**, con cui ai processi viene assegnata la CPU nell'ordine in cui l'hanno richiesta. Fondamentalmente, c'è una sola coda di processi nello stato di pronto, e quando il primo job, di mattina, arriva al sistema, inizia immediatamente e può stare in esecuzione quanto a lungo vuole. Come arrivano altri job, questi vengono messi in coda; quando il processo in esecuzione si blocca, il primo processo in coda è il prossimo processo che viene eseguito; e in seguito, quando il processo bloccato passa allo stato di pronto, viene messo in fondo alla coda come un job appena arrivato.

La grande forza di questo algoritmo è che è facile da capire e ugualmente facile da programmare; è anche equo, tanto quanto è equo assegnare pochi biglietti per un incontro sportivo o per un concerto a persone che sono intenzionate a stare in fila a partire dalle 2 del mattino. In questo algoritmo, una lista concatenata tiene traccia di tutti i processi nello stato di pronto; scegliere un processo da eseguire semplicemente richiede di rimuoverlo dalla testa della coda, mentre aggiungere un nuovo job o un processo non

più bloccato significa semplicemente porlo alla fine della coda. Cosa potrebbe essere più semplice?

Sfortunatamente, l'algoritmo primo arrivato, primo servito ha anche un grosso svantaggio: supponete che ci sia un processo compute bound che viene eseguito per 1 secondo alla volta e molti processi I/O bound che usano poco tempo di CPU, ognuno dei quali deve eseguire 1000 letture da disco per terminare. Il processo compute bound viene eseguito per 1 secondo, quindi legge un blocco da disco, poi tutti i processi che fanno ingresso/uscita vengono eseguiti ed iniziano le letture da disco. Quando il processo compute bound ottiene il suo blocco di disco, viene eseguito per un altro secondo, seguito da tutti i processi I/O bound in successione veloce.

Il risultato finale è che ogni processo I/O bound ottiene di leggere 1 blocco al secondo e impiegherà 1000 secondi per finire. Con un algoritmo di schedulazione che prerilascia il processo CPU bound ogni 10 millisecondi, i processi I/O bound termineranno in 10 secondi anziché in 1000, e senza rallentare molto il processo compute bound.

Shortest Job First

Adesso diamo un'occhiata a un altro algoritmo batch senza prerilascio che ha come presupposto la conoscenza anticipata dei tempi di esecuzione. In una compagnia di assicurazioni, per esempio, si può predire con buona approssimazione quanto tempo durerà l'esecuzione di un lotto di 1000 richieste, dal momento che job simili vengono svolti ogni giorno. Quando nella coda di ingresso si trovano una serie di job, tutti di uguale importanza, in attesa di essere eseguiti, lo scheduler dovrebbe usare lo **shortest job first** (cioè, il job più breve viene servito per primo). Guardiamo la Figura 2.39: qui troviamo quattro job, A, B, C, e D, con tempi di esecuzione rispettivamente di 8, 4, 4, e 4 minuti. Mandandoli in esecuzione in quest'ordine, il tempo di turnaround per A è di 8 minuti, per B è di 12 minuti, per C è di 16 minuti e per D è di 20 minuti, con una media di 14 minuti.

Adesso consideriamo una schedulazione shortest job first di questi job, come quella di Figura 2.39(b): i tempi di turnaround sono ora di 4, 8, 12 e 20 minuti, con una media di 11 minuti. Si può anche dimostrare che la politica shortest job first è ottima: si consideri il caso di quattro job, con tempi di esecuzione rispettivamente a , b , c e d : il primo job finisce al tempo a , il secondo finisce al tempo $a + b$ e così via. Il tempo medio di turnaround è di $(4a + 3b + 2c + d)/4$: è chiaro che a contribuisce alla media di più degli altri tempi, quindi dovrebbe essere il job più corto, con b subito dopo, quindi c e infine d che è il job più

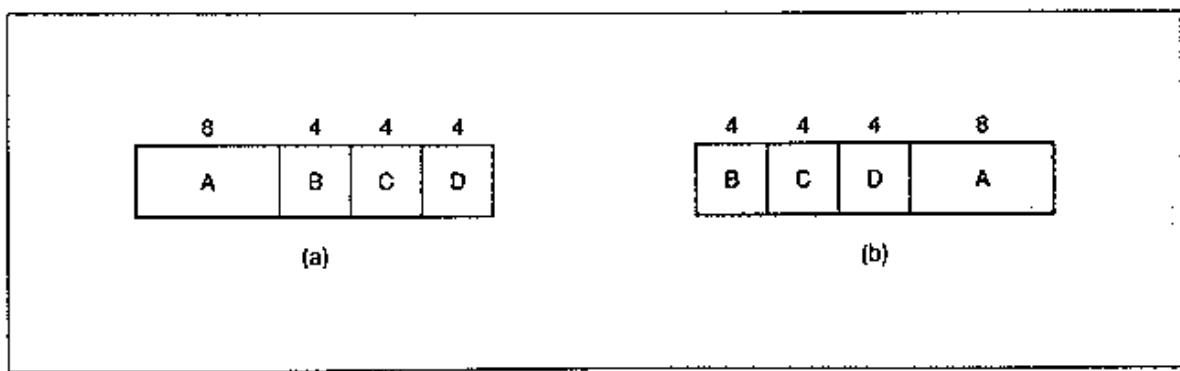


Figura 2.39 Un esempio di schedulazione shortest job first. (a) Esecuzione dei quattro job nell'ordine originale. (b) Loro esecuzione nell'ordine shortest job first.

lungo, dal momento che influenza solo il proprio tempo di turnaround. Lo stesso ragionamento può essere applicato ad un numero arbitrario di job.

Val la pena di notare che l'algoritmo shortest job first è ottimo solo nel caso in cui tutti i job siano disponibili contemporaneamente; come controesempio, considerate 5 job, con nomi da A a E e con tempi di esecuzione rispettivamente di 2, 4, 1, 1, e 1; i loro tempi di arrivo siano 0, 0, 3, 3 e 3. Inizialmente, possono essere scelti solo A e B, dal momento che gli altri job non sono ancora arrivati; usando una politica shortest job first, questi job vengono mandati in esecuzione nell'ordine: A, B, C, D, E, per un tempo medio di attesa di 4,6. Invece, mandandoli in esecuzione nell'ordine B, C, D, E, A, si ottiene un tempo medio di attesa di 4,4.

Shortest Remaining Time Next

Una versione con prerilascio dello shortest job first è lo **shortest remaining time next** (cioè, viene eseguito per primo il processo con il tempo rimasto più breve). Con questo algoritmo, lo schedulatore sceglie sempre il processo il cui tempo di esecuzione residuo è il più breve: anche per questo algoritmo, il tempo di esecuzione deve essere noto in anticipo. Quando arriva un nuovo job, il suo tempo totale viene confrontato con il tempo residuo del processo corrente; se il nuovo job ha bisogno di meno tempo per finire del processo corrente, questo viene sospeso e viene iniziato il nuovo job. Questo schema permette a nuovi job brevi di ottenere un buon servizio.

Schedulazione a tre livelli

Da una certa prospettiva, i sistemi batch permettono di schedulare a tre diversi livelli, come illustrato in Figura 2.40: come i job arrivano al sistema, vengono inizialmente posti in una coda d'ingresso, memorizzata su disco. Lo **schedulatore d'ingresso** decide quale job deve essere ammesso al sistema, gli altri vengono tenuti nella coda d'ingresso finché non vengono selezionati. Un algoritmo tipico per il controllo d'ingresso dovrebbe cercare un miscuglio di job compute bound e I/O bound; in alternativa, potrebbe velocemente ammettere i job brevi, mentre job più lunghi dovrebbero aspettare. Lo **schedulatore d'ingresso** è libero di tenere qualche job nella coda d'ingresso e ammettere job che arrivano dopo, se così decide.

Una volta che un job è stato ammesso nel sistema, si crea un processo per tale job, che può entrare in competizione per la CPU. Se invece capita che il numero di processi sia così grande che non c'è abbastanza spazio per tutti in memoria, alcuni processi devono essere spostati dalla memoria al disco. Il secondo livello di schedulazione si occupa di decidere quali processi debbano essere tenuti in memoria e quali vadano tenuti su disco: chiameremo questo **schedulatore di memoria**, dal momento che determina quali processi vadano tenuti in memoria e quali su disco.

Questa decisione deve essere frequentemente sottoposta a revisione per permettere ai processi su disco di ottenere qualche servizio. Comunque, dal momento che portare e prelevare un processo su e da disco è un'operazione costosa, la revisione probabilmente non dovrebbe accadere più spesso di una volta per secondo, forse meno spesso. Se il contenuto della memoria principale viene cambiato troppo spesso, gran parte della larghezza di banda del disco verrà sprecata, rallentando le operazioni di ingresso/uscita dei file.

Per ottimizzare nel complesso le prestazioni del sistema, lo **schedulatore di memoria** potrebbe voler decidere attentamente quanti processi vuole in memoria, cioè qual è il gra-

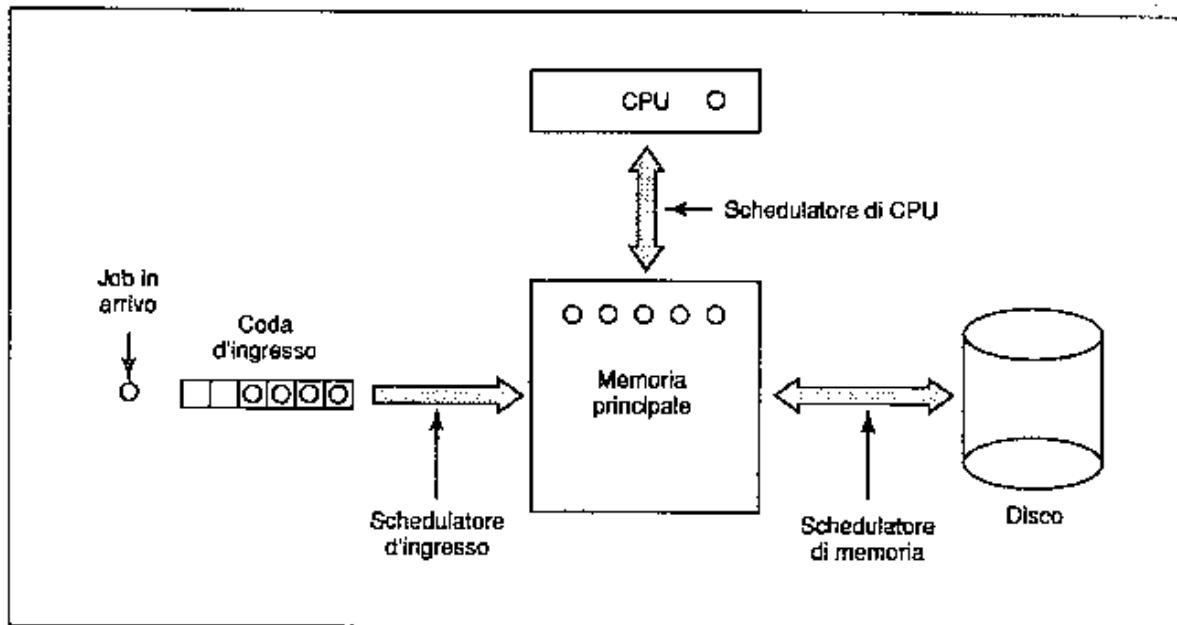


Figura 2.40 Schedulazione a tre livelli.

do di multiprogrammazione, e quale tipo di processi. Se ha informazioni su quali processi sono compute bound e quali sono I/O bound, può provare a tenere un miscuglio di questi tipi di processi in memoria. Come approssimazione molto rozza, se una certa classe di processi esegue calcoli per circa il 20% del tempo, tenerne cinque in giro è approssimativamente il numero giusto per tenere la CPU occupata. Daremo un'occhiata a un modello di multiprogrammazione leggermente migliore nel Capitolo 4.

Per prendere le sue decisioni, lo schedulatore di memoria periodicamente rivede ogni processo su disco per decidere se portarlo o meno in memoria. Fra i criteri che può usare per prendere le sue decisioni ci sono i seguenti:

1. Quanto tempo è passato da quando il processo è stato scaricato sul/dal disco?
2. Quanto tempo di CPU ha avuto il processo recentemente?
3. Quanto è grande il processo? (I più piccoli non danno fastidio.)
4. Quanto è importante il processo?

Il terzo livello di schedulazione sceglie realmente uno dei processi nello stato di pronto, che si trovano nella memoria principale, perché venga eseguito per primo; spesso è chiamato **schedulatore di CPU** ed è quello che generalmente si intende parlando dello "schedulatore". Qui si può usare ogni algoritmo adatto, con prerilascio o senza, compresi quelli descritti sopra, oltre agli algoritmi che saranno descritti nella prossima sezione.

2.5.3 Schedulazione nei sistemi interattivi

Daremo ora un'occhiata ad alcuni algoritmi che possono essere usati nei sistemi interattivi, e che possono essere usati altrettanto bene anche come schedulatori di CPU nei sistemi batch. Mentre la schedulazione a tre livelli qui non è possibile, la schedulazione a due livelli (schedulatore di memoria e schedulatore di CPU) è possibile e comune. Nel seguito ci focalizzeremo sulla schedulazione di CPU.

Schedulazione round robin

Occupiamoci ora di alcuni algoritmi di schedulazione specifici; uno degli algoritmi più vecchi, più semplici, più imparziali e più usati è il **round robin**: ad ogni processo viene assegnato un intervallo di tempo, chiamato il suo **quanto**, durante il quale può essere in esecuzione. Se alla fine del quanto il processo è ancora in esecuzione, la CPU viene rilasciata ed assegnata ad un altro processo; se il processo si è bloccato o è terminato prima che sia trascorso il quanto, l'assegnazione della CPU viene ovviamente eseguita quando il processo si blocca. Il round robin è facile da implementare; lo scheduler ha bisogno solo di mantenere una lista di processi eseguibili, come illustrato in Figura 2.41(a); e quando scade il quanto, il processo viene messo in fondo alla lista, come mostrato in Figura 2.41 (b).

L'unica questione interessante nel round robin è la durata del quanto: il passaggio da un processo ad un altro richiede una certa quantità di tempo per gli aspetti amministrativi: salvare e caricare registri e mappe di memoria, aggiornare varie tabelle e liste, svuotare e ricaricare la cache, eccetera. Supponiamo che questo scambio di processi, o **cambio di contesto**, come viene talvolta chiamato, impieghi 1 millisecondo, compreso il cambio delle mappe di memoria, svuotare e ricaricare la cache, eccetera, e che il quanto sia di 4 millisecondi. Con questi parametri, dopo aver svolto 4 millisecondi di lavoro utile, la CPU dovrà spendere 1 millisecondo per lo scambio di processi: il venti per cento del tempo di CPU risulterà sprecato in overhead di gestione, e chiaramente è troppo.

Per migliorare l'efficienza della CPU potremmo mettere il quanto a, diciamo, 100 millisecondi: ora il tempo sprecato è solo l'un per cento. Ma consideriamo cosa succede se 10 utenti interattivi battono il tasto di invio circa nello stesso istante: dieci processi verranno messi nella lista dei processi eseguibili. Se la CPU è inattiva, il primo comincerà immediatamente, il secondo non potrà cominciare che dopo 100 millisecondi, e così via; l'ultimo sfortunato potrebbe dover attendere 1 secondo prima di avere una possibilità, se tutti gli altri usano completamente il loro quanto. La maggior parte degli utenti considererebbe fiacco un tempo di risposta di 1 secondo per un comando breve.

Un altro fattore è che se il quanto è impostato ad un tempo più lungo del burst di CPU medio, la prelazione verrà effettuata di rado, mentre la maggior parte dei processi eseguirà un'operazione che li blocca prima che il quanto scada, causando uno scambio di processo. Eliminare la prelazione migliora le prestazioni, perché gli scambi di processi si verificano solo quando sono logicamente necessari, cioè quando un processo si blocca e non può più continuare.

La conclusione può essere formulata come segue: assegnare un quanto troppo breve provoca troppi cambi di contesto e peggiora l'efficienza della CPU, ma assegnarlo troppo lungo può provocare tempi di risposta lunghi per richieste interattive brevi. Un quanto di circa 20-50 millisecondi è spesso un compromesso ragionevole.

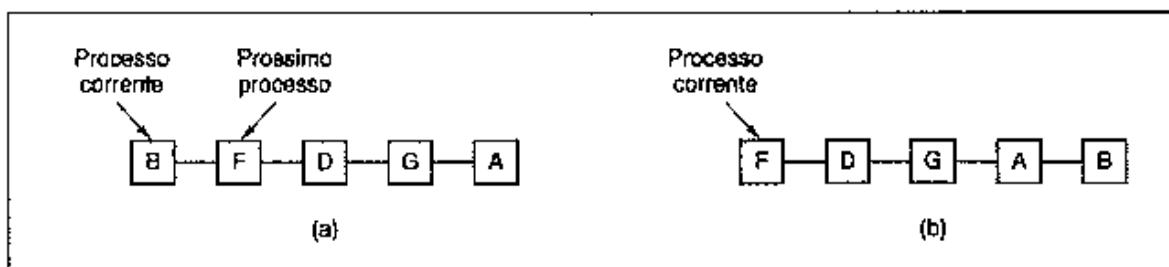


Figura 2.41 Schedulazione round robin. [a] La lista di processi eseguibili. [b] La lista di processi eseguibili dopo che è scaduto il quanto di B.

Schedulazione con priorità

La schedulazione round robin ipotizza implicitamente che tutti i processi abbiano la stessa importanza, ma spesso le persone che possiedono e gestiscono calcolatori multiutente hanno idee diverse su questo argomento: in un'università, l'ordine gerarchico potrebbe essere prima i presidi, poi i professori, segretari, bidelli ed infine gli studenti. La necessità di prendere in considerazione fattori esterni porta alla **schedulazione con priorità**, la cui idea base è banale: ad ogni processo viene assegnata una priorità e viene concessa l'esecuzione al processo eseguibile con priorità più alta.

Persino su un PC con un solo utente possono esserci più processi, alcuni più importanti degli altri. Ad esempio, ad un processo demone che invia la posta elettronica in background potrebbe essere assegnata una priorità più bassa rispetto ad un processo che mostra in tempo reale un film sullo schermo.

Per evitare che i processi ad alta priorità rimangano indefinitamente in esecuzione, lo scheduler può diminuire la priorità del processo in esecuzione ad ogni tic (cioè ad ogni interruzione) del clock e, se questa azione fa scendere la priorità del processo sotto quella del processo successivo a priorità più alta, si verifica un cambio di contesto. In alternativa, ad ogni processo può essere assegnato un quanto di tempo massimo per il quale gli è permesso rimanere in esecuzione e, quando il quanto scade, viene data la possibilità di andare in esecuzione al processo successivo a priorità più alta.

Le priorità possono essere assegnate ai processi staticamente o dinamicamente; in un calcolatore dell'esercito, i processi lanciati dai generali potrebbero cominciare a priorità 100, i processi lanciati dai colonnelli a 90, quelli dei maggiori a 80, dei capitani a 70, dei tenenti a 60 e così via. In alternativa, in un centro di calcolo commerciale, i job ad alta priorità potrebbero costare 100 dollari all'ora, quelli a media priorità 75 dollari all'ora e quelli a bassa priorità 50 dollari all'ora. Il sistema UNIX ha un comando, *nice*, che permette ad un utente di ridurre volontariamente la priorità del suo processo, per essere gentile con gli altri utenti: nessuno lo usa mai.

Le priorità possono anche essere assegnate dinamicamente dal sistema per perseguire certi obiettivi di sistema; ad esempio, alcuni processi sono molto legati all'ingresso/uscita e passano quindi la maggior parte del loro tempo in attesa del completamento delle operazioni di ingresso/uscita; ogni volta che un processo del genere vuole la CPU gliela si dovrebbe assegnare immediatamente, per permettergli di avviare la sua prossima richiesta di ingresso/uscita, che può poi procedere in parallelo con il vero calcolo di un altro processo. Fare in modo che un processo I/O bound aspetti a lungo la CPU significherebbe solo lasciargli occupare la memoria per un tempo inutilmente lungo. Un algoritmo semplice, che rende un buon servizio ai processi I/O bound è assegnare priorità $1/f$, dove f è la frazione di quanto che un processo ha usato: un processo che usa solo 1 millisecondo del suo quanto di 50 millisecondi otterebbe priorità 50, mentre un processo che viene eseguito per 25 millisecondi prima di bloccarsi otterebbe priorità 2, e un processo che usa l'intero quanto di tempo avrebbe priorità 1.

Spesso è conveniente raggruppare i processi in classi di priorità e usare la schedulazione a priorità fra le classi, con schedulazione round robin all'interno di ciascuna classe. La Figura 2.42 mostra un sistema con quattro classi di priorità: l'algoritmo di schedulazione è il seguente: fino a che esistono processi eseguibili nella classe di priorità 4, manda in esecuzione ciascuno di essi per un quanto, con schedulazione round robin e senza preoccuparsi delle classi di priorità più bassa; se la classe di priorità 4 è vuota, esegui i processi della classe 3 in round robin; se le classi 4 e 3 sono entrambe vuote, allora esegui i processi della classe 2 in round robin, e così via. Se le priorità non vengono aggiornate di tanto in tanto, le classi a priorità più bassa possono languire all'infinito.

Code multiple

Uno dei primi scheduleri a priorità si trova in CTSS (Corbató e altri., 1962). CTSS aveva il problema che il cambio di contesto fra processi era molto lento, perché il 7094 poteva mantenere in memoria un solo processo; ciascun cambio di contesto significava scaricare il processo corrente sul disco e da qui caricarne uno nuovo. I progettisti di CTSS capirono molto in fretta che era più efficiente dare ai processi CPU bound un grande quanto di tempo una volta ogni tanto, piuttosto che dar loro piccoli quanti più di frequente (al fine di ridurre lo swapping su disco). D'altra parte, dare a tutti i processi un quanto di tempo grande significa alzare il tempo di risposta, come abbiamo già visto. La loro soluzione fu quella di introdurre classi di priorità: i processi della classe più alta venivano eseguiti per un quanto, quelli della classe immediatamente più bassa per due quanti, i processi nella classe successiva per quattro quanti e così via. Ogni qualvolta un processo rimaneva in esecuzione per tutto il proprio quanto di tempo, era spostato in basso di una classe.

Come esempio, consideriamo un processo che abbia necessità di usare la CPU per 100 quanti: inizialmente gli viene assegnato un quanto, trascorso il quale viene scaricato su disco; la volta successiva, gli vengono assegnati due quanti, prima di essere scaricato su disco. Ai passi successivi ottiene 4, 8, 16, 32 e 64 quanti, sebbene all'ultimo passo il processo ne usi, per completare il proprio lavoro, solo 37. In questo modo sono necessari solo 7 scaricamenti su disco (compreso il caricamento iniziale) invece dei 100 necessari con l'algoritmo di round robin puro. Inoltre, man mano che un processo sprofonda sempre più nelle code di priorità, viene mandato in esecuzione con sempre minor frequenza, risparmiando così tempo di CPU per l'esecuzione dei processi interattivi corti.

Per impedire che un processo che, appena partito, ha bisogno di essere eseguito a lungo e, in seguito, diventa interattivo, venga punito per sempre, viene adottata la seguente politica. Ogni volta che viene battuto al terminale il tasto INVIO, il processo appartenente a quel terminale viene spostato nella classe a priorità più alta, nell'ipotesi che stia per diventare interattivo. Un bel giorno, un utente con un processo pesantemente CPU bound scoprì che sedersi al terminale e battere il tasto INVIO a caso ogni pochi secondi faceva meraviglie per il suo tempo di risposta, e lo disse a tutti i suoi amici. Morale della storia: far le cose giuste in pratica è molto più difficile che farle giuste in linea di principio.

Per assegnare i processi alle classi di priorità sono stati utilizzati molti altri algoritmi; ad esempio, il celebre sistema XDS 940 (Lampson, 1968), realizzato a Berkeley, aveva quattro classi di priorità chiamate terminale, ingresso/uscita, quanto breve e quanto lungo. Quando un processo in attesa di un dato in ingresso da un terminale alla fine veniva

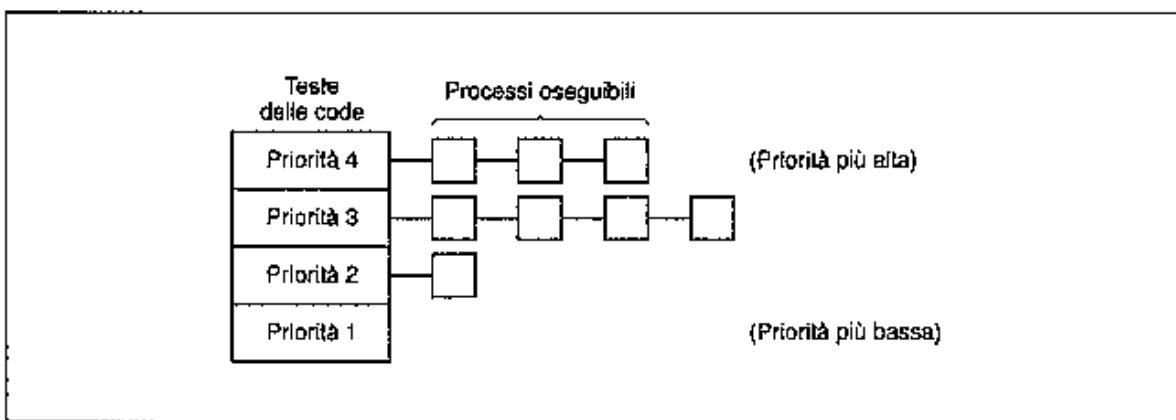


Figura 2.42 Un algoritmo di schedulazione con quattro classi di priorità.

risvegliato, finiva nella classe a priorità più alta (classe terminale); quando un processo in attesa della disponibilità di un blocco da disco diveniva pronto, passava nella seconda classe; quando un processo era ancora in esecuzione allo scadere del suo quanto di tempo, veniva inizialmente piazzato nella terza classe; in ogni caso, se un processo usava tutto il suo quanto di tempo per molte volte di seguito, senza bloccarsi per il terminale o per altre attività di ingresso/uscita, veniva posto nella coda più bassa. Molti altri sistemi usano qualcosa di simile per favorire gli utenti e i processi interattivi rispetto a quelli in background.

Shortest process next

Dal momento che lo shortest job first produce sempre, nei sistemi batch, il minimo tempo di risposta medio, sarebbe carino se potesse essere usato anche per i processi interattivi: in una certa misura, può esserlo. I processi interattivi seguono di solito lo schema in cui aspettano un comando, lo eseguono, ne aspettano un altro, lo eseguono, eccetera. Se consideriamo l'esecuzione di ciascun comando come un "job" a sé stante, potremmo minimizzare il tempo totale di risposta usando la schedulazione shortest job first. Il solo problema sarebbe quello di determinare quale dei processi correntemente eseguibili è il più breve.

Un approccio è quello di fare delle stime in base al comportamento passato ed eseguire il processo con il tempo di esecuzione che è stato stimato più breve. Supponiamo che il tempo stimato per l'esecuzione di un comando su un certo terminale sia T_0 e che la sua successiva esecuzione richieda un tempo pari a T_1 . Potremmo aggiornare la nostra stima prendendo la media pesata di questi due valori e cioè $aT_0 + (1 - a)T_1$; mediante la scelta di a possiamo decidere se avere una stima che dimentica in fretta il comportamento passato o lo ricorda per un lungo tempo. Con $a = 1/2$, abbiamo stime successive pari a:

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Dopo tre nuove esecuzioni, il peso di T_0 nella nuova stima è ridotto ad 1/8.

Questa tecnica di stimare il prossimo valore di una serie prendendo la media pesata del valore appena misurato e del valore precedente, viene chiamata **invecchiamento (aging)** ed è applicabile a molte situazioni in cui la previsione va fatta basandosi su valori precedenti. L'invecchiamento è particolarmente semplice da implementare quando $a = 1/2$, perché tutto ciò che occorre fare è aggiungere il nuovo valore alla stima corrente e dividere la somma per 2 (con uno shift a destra di 1 bit).

Schedulazione garantita

Un approccio completamente diverso è quello di fare all'utente delle vere e proprie promesse relative alla prestazioni e, successivamente, mantenerle. Una promessa realistica da fare e facile da mantenere è la seguente: se ci sono n utenti collegati mentre voi state lavorando, riceverete $1/n$ della potenza della CPU. In modo simile, su un sistema a singolo utente con n processi equivalenti in esecuzione, ciascuno dovrebbe ottenere $1/n$ dei cicli di CPU.

Per mantenere la promessa, il sistema deve tener traccia di quanta CPU ogni processo ha ottenuto dalla sua creazione, quindi calcola la quantità di CPU a cui ciascuno ha diritto, dividendo il tempo trascorso dalla creazione per n . Dal momento che il tempo di CPU che ciascun processo ha veramente avuto è noto, è facile calcolare il rapporto fra il tem-

po di CPU utilizzato e quello a cui ha diritto. Un rapporto di 0,5 significa che il processo ha avuto solo metà del tempo che gli spettava; un rapporto di 2,0 significa che il processo ha avuto il doppio del tempo che gli sarebbe spettato. L'algoritmo stabilisce di mandare in esecuzione il processo che ha il rapporto più basso, fintanto che il suo rapporto non salé al di sopra del rapporto del suo avversario più vicino.

Rapporto di tempo in CPU < tempo di pre-emptione

Schedulazione a lotteria

Sebbene fare promesse agli utenti e mantenerle è una buona idea, è difficile da implementare. Comunque, può essere usato un altro algoritmo per fornire risultati prevedibili e simili, con un'implementazione molto più semplice, chiamato **schedulazione a lotteria** (Waldspurger e Weihl, 1994).

L'idea di base è fornire ai processi biglietti della lotteria per varie risorse del sistema, come il tempo di CPU. Ognqualvolta deve essere presa una decisione di schedulazione, viene scelto a caso un biglietto della lotteria e il processo che lo possiede ottiene la risorsa. Per la schedulazione della CPU, il sistema potrebbe tenere una lotteria 50 volte al secondo, con ogni vincitore che ottiene 20 millisecondi di tempo di CPU come premio.

Per parafrasare George Orwell: "Tutti i processi sono uguali, ma qualche processo è più uguale degli altri"; ai processi più importanti possono essere assegnati biglietti extra, per aumentare la loro probabilità di vincere. Se ci sono 100 biglietti in circolazione, ed un processo ne possiede 20, avrà il 20% di possibilità di vincere ogni lotteria, ed alla lunga, otterrà circa il 20% della CPU. A differenza dello schedulatore a priorità, dove è molto difficile stabilire cosa significhi veramente avere una priorità pari a 40, qui la regola è chiara: un processo che detiene una frazione f dei biglietti avrà circa una frazione f della risorsa in questione.

La schedulazione a lotteria ha diverse proprietà interessanti: ad esempio, se arriva un nuovo processo, e gli viene concesso qualche biglietto, alla successiva lotteria avrà una possibilità di vincere in proporzione al numero di biglietti che possiede; in altre parole, la schedulazione a lotteria risponde molto bene alle aspettative.

I processi che cooperano possono scambiarsi biglietti se lo desiderano; ad esempio, quando un processo client invia un messaggio a un processo server e poi si blocca, può dare tutti i suoi biglietti al server, per aumentare le possibilità del server di essere eseguito per successivo. Quando il server ha finito, restituisce i biglietti in modo che il cliente possa essere eseguito di nuovo; in realtà, in assenza di clienti, i server non hanno affatto bisogno di biglietti.

La schedulazione a lotteria può essere usata per risolvere problemi che sono difficili da gestire con altri metodi; un esempio è un server video nel quale diversi processi passano delle sequenze video ai loro clienti, ma a diverse frequenze di fotogrammi. Supponete che i processi abbiano bisogno di frame a 10, 20 e 25 frame al secondo: allocando a questi processi rispettivamente 10, 20 e 25 biglietti si divideranno automaticamente la CPU circa nelle giuste proporzioni, cioè 10 : 20 : 25.

Schedulazione fair-share (porzioni equi)

Finora abbiamo supposto che ogni processo sia schedulato per se stesso, senza considerare a chi appartiene; come risultato, se l'utente 1 inizia 9 processi e l'utente 2 inizia un processo, con schedulazione round robin o priorità uguali, l'utente 1 avrà il 90% della CPU e l'utente 2 ne avrà solo il 10%.

Per evitare questa situazione, alcuni sistemi prendono in considerazione chi è il pro-

prietario di un processo prima di schedularlo: in questo modello, ad ogni utente viene assegnata una frazione della CPU e lo scheduler sceglie i processi in modo da rispettarla, così, se a due utenti è stato assicurato il 50% della CPU, otterranno questo, non importa quanti processi hanno creato.

Come esempio, considerate un sistema con due utenti, a ciascuno dei quali è stato assicurato il 50% della CPU. L'utente 1 ha quattro processi, A, B, C e D e l'utente 2 ha un solo processo, E. Se viene utilizzata la schedulazione round robin, una possibile sequenza di schedulazione che rispetta tutti i vincoli è questa:

A B B E C E D E A E B E C E D E ...

D'altro canto, se l'utente 1 ha diritto al doppio del tempo di CPU rispetto all'utente 2, potremmo avere:

A B E C D E A B E C D E ...

Certamente esistono molte altre possibilità, e possono essere sfruttate, a seconda di qual è la nozione di equità.

~~2.5.4 Schedulazione nei sistemi real time~~

In un sistema **real time** (in tempo reale) il tempo gioca un ruolo essenziale. Di solito, uno o più dispositivi esterni al calcolatore generano impulsi, ai quali il calcolatore deve reagire in modo appropriato entro una quantità di tempo fissata. Ad esempio, il calcolatore in un riproduttore di compact disc prende i bit che escono dal drive, e deve convertirli in musica in un intervallo di tempo molto ristretto; se il calcolo richiede troppo tempo, la musica suonerà strana. Altri sistemi real time sono il monitoraggio dei pazienti in un'unità ospedaliera di cure intensive, il pilota automatico di un aereo e il controllo robotizzato in una fabbrica automatizzata. In tutti questi casi, avere la risposta corretta, ma in ritardo è come non averla affatto, se non peggio.

I sistemi real time sono normalmente classificati in **hard real time**, cioè sistemi che hanno scadenze incondizionate da rispettare, e in **soft real time**, intendendo che non rispettare occasionalmente una scadenza non è opportuno, ma tuttavia è tollerato. In entrambi i casi, il comportamento real time viene raggiunto dividendo il programma in processi, il cui comportamento è prevedibile e noto in anticipo. Questi processi hanno normalmente una vita breve e possono essere eseguiti nella loro completezza in molto meno di un secondo. Quando viene rilevato un evento esterno, è compito dello scheduler schedulare i processi in modo che tutte le scadenze vengano rispettate.

Gli eventi a cui un sistema real time può dover reagire possono essere ulteriormente classificati in **periodici** (che si verificano ad intervalli di tempo regolari) o **aperiodici** (che si verificano in modo imprevedibile). Un sistema può dover reagire a molte serie di eventi periodici, a seconda di quanto tempo richiede ciascun evento per eseguire i processi, può persino non essere possibile gestirli tutti. Ad esempio, se ci sono m eventi periodici e se l'evento i arriva con periodo P_i e richiede C_i secondi di tempo di CPU per essere gestito, il carico può essere gestito solo se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Un sistema real time che rispetta questo criterio è detto **schedulabile**.

Come esempio, considerate un sistema soft real time con tre eventi periodici, con periodi rispettivamente di 100, 200 e 500 millisecondi. Se questi eventi richiedono rispettivamente 50, 30 e 100 millisecondi di tempo di CPU per evento, il sistema è schedulabile perché $0,5 + 0,15 + 0,2 < 1$. Se viene aggiunto un quarto evento, con periodo di 1 secondo, il sistema rimarrà schedulabile finché questo evento non richiederà più di 150 millisecondi di tempo di CPU per evento. In questo calcolo si suppone implicitamente che l'overhead per il cambio di contesto sia così piccolo da poter essere ignorato.

Gli algoritmi di schedulazione real time possono essere statici o dinamici: i primi prendono le decisioni di schedulazione prima che il sistema inizi l'esecuzione, mentre i secondi le prendono a tempo di esecuzione. La schedulazione statica funziona solo quando sono disponibili in anticipo le informazioni complete circa il lavoro da fare e le scadenze da rispettare, mentre gli algoritmi di schedulazione dinamica non hanno queste restrizioni. Rimandiamo il nostro studio degli algoritmi specifici a quando tratteremo i sistemi multimediali nel Capitolo 7.

2.5.5 Politiche e meccanismi

Fino ad ora abbiamo tacitamente supposto che tutti i processi in un sistema appartengano a utenti diversi e quindi competano per l'uso della CPU; sebbene questo spesso sia vero, a volte accade che un processo abbia molti figli in esecuzione sotto il suo controllo. Ad esempio, un sistema di gestione di basi di dati può avere molti figli, ciascuno dei quali potrebbe lavorare su una richiesta diversa, oppure potrebbe avere una qualche funzione specifica da svolgere (interpretazione delle richieste, accesso al disco, eccetera). È possibile che il processo principale abbia una buona idea di quale, fra tutti i figli, sia il più importante, o il più critico rispetto al tempo, e di quale lo sia meno. Sfortunatamente, nessuno degli scheduler visti fino ad ora accetta dati in ingresso dai processi utente, da utilizzare per le decisioni di schedulazione; di conseguenza, lo scheduler effettua di rado la scelta migliore.

La soluzione a questo problema è quella di separare il **meccanismo di schedulazione** dalla **politica di schedulazione**, il che significa che l'algoritmo di schedulazione viene reso in qualche modo parametrico, ma i parametri possono essere forniti dai processi utente. Consideriamo di nuovo il caso del sistema di gestione per basi di dati e supponiamo che il kernel usi uno scheduler a priorità, ma metta a disposizione una chiamata di sistema con la quale un processo possa assegnare (e modificare) la priorità dei propri figli. In questo modo il genitore può controllare in modo dettagliato come vengono schedulati i suoi figli, anche se egli non esegue la schedulazione: il meccanismo sta nel kernel ma la politica è stabilita da un processo utente.

2.5.6 Schedulazione dei thread

Quando diversi processi hanno thread multipli, abbiamo la presenza contemporanea di due livelli di parallelismo: processi e thread. In sistemi di questo tipo la schedulazione differisce sostanzialmente a seconda che siano supportati thread di livello utente o thread di livello kernel (o entrambi).

Consideriamo prima i thread di livello utente: dal momento che il kernel non è consapevole dell'esistenza dei thread, opera come sempre, scegliendo un processo, diciamo *A*, e dando ad *A* il controllo per il suo quanto. Lo scheduler dei thread in *A* decide quale thread eseguire, diciamo *A1*. Dal momento che non ci sono interruzioni di clock per

multiprogrammare i thread, questo thread può continuare la sua esecuzione per quanto vuole. Se esaurisce completamente il suo quanto, il kernel selezionerà un altro processo da eseguire.

Quando, alla fine, il processo A torna in esecuzione, il thread A1 verrà rimandato in esecuzione e continuerà a consumare tutto il tempo di A finché non ha finito. Comunque, il suo comportamento antisociale non ha conseguenze sugli altri processi, che otterranno le porzioni che lo schedulatore considera appropriate, qualunque cosa succeda all'interno del processo A.

Ora considerate il caso in cui i thread di A abbiano un lavoro relativamente piccolo da svolgere per ogni burst di CPU, ad esempio 5 millisecondi di lavoro in un quanto di 50 millisecondi; di conseguenza, ciascuno viene eseguito per un breve periodo, quindi restituisce la CPU allo schedulatore dei thread. Questo potrebbe portare alla sequenza A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, prima che il kernel passi al processo B. Questa situazione è illustrata in Figura 2.43(a).

L'algoritmo di schedulazione usato dal sistema a tempo di esecuzione è uno di quelli descritti sopra; in pratica, la schedulazione round robin e la schedulazione a priorità sono i più comuni. Il solo vincolo è l'assenza di un clock per interrompere un thread che è stato eseguito per troppo tempo.

Considerate ora la situazione con i thread a livello kernel: qui il kernel sceglie un thread particolare da eseguire; non deve tenere in considerazione a quale processo appartiene, ma, se vuole, può farlo. Al thread è assegnato un quanto e viene forzatamente sospeso se lo supera; con un quanto di 50 millisecondi una thread che si bloccano dopo 5 millisecondi, l'ordine dei thread per un periodo di 30 millisecondi potrebbe essere A1, B1, A2, B2, A3, B3, cosa che non sarebbe possibile con questi parametri e thread di livello utente. Questa situazione è parzialmente raffigurata in Figura 2.43(b).

Una differenza importante tra thread di livello utente e thread di livello kernel sono le prestazioni; fare un cambio di contesto fra thread di livello utente richiede una manciata di istruzioni macchina, mentre con i thread di livello kernel richiede un completo cambio

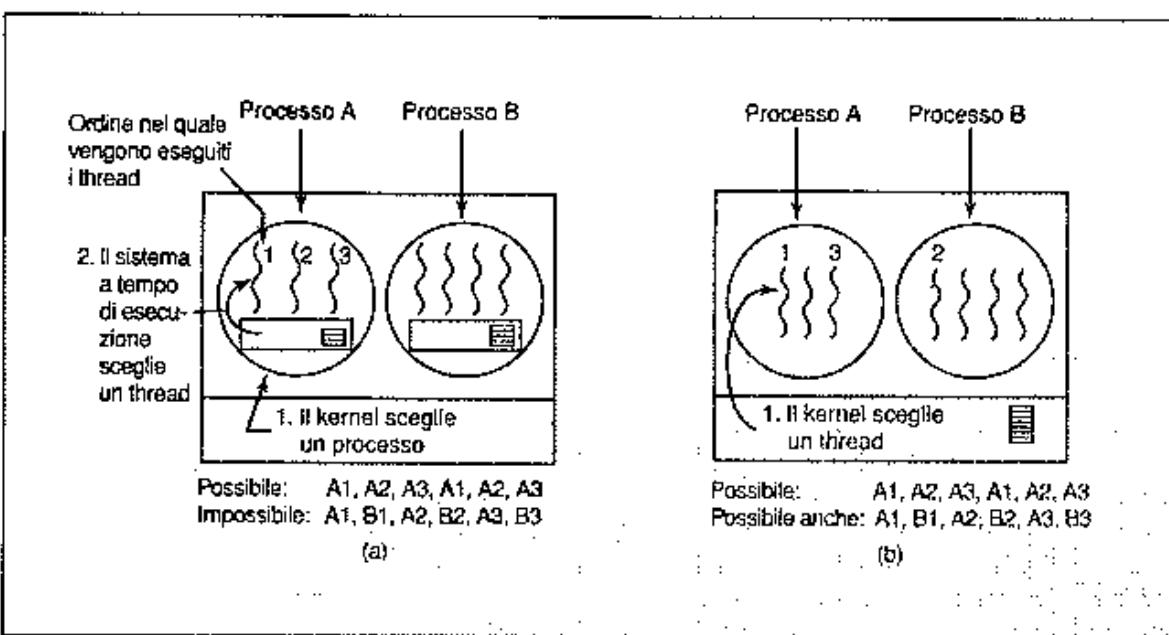


Figura 2.43 (a) Una possibile schedulazione di thread di livello utente con un quanto di processo di 50 millisecondi e thread che vengono eseguiti con burst di CPU di 5 millisecondi. (b) Una schedulazione possibile di thread di livello kernel con le stesse caratteristiche del caso (a).

di contesto, cambiando la mappa di memoria e invalidando la cache, quindi è diversi ordini di grandezza più lento. D'altro canto, con i thread di livello utente, avere un thread bloccato su un'operazione di ingresso/uscita non sospende l'intero processo come accade con i thread di livello utente.

Dal momento che il kernel sa che il cambio da un thread nel processo A a un thread del processo B è più costoso che non eseguire un secondo thread nel processo A (a causa del cambio della mappa di memoria e dell'annullamento del contenuto della cache), può prendere in considerazione quest'informazione quando prende una decisione. Ad esempio, dati due thread che sarebbero altrimenti equivalenti, uno appartenente al processo in cui un thread si è appena bloccato, ed uno appartenente ad un processo diverso, si potrebbe dare la precedenza al primo.

Un altro fattore importante è che i thread di livello utente possono utilizzare uno scheduler di thread specifico per l'applicazione: consideriamo, per esempio, il server Web di Figura 2.10, e supponete che un thread lavoratore si sia appena bloccato e che il thread dispatcher e due thread lavoratori siano nello stato di pronto. Quale dovrebbe essere il prossimo? Il sistema a tempo di esecuzione, sapendo cosa fanno tutti i thread, può facilmente scegliere di eseguire il dispatcher, così che possa far partire un altro lavoratore. Questa strategia massimizza la quantità di parallelismo in un ambiente dove i lavoratori si bloccano di frequente sulle operazioni di ingresso/uscita su disco. Con i thread di livello kernel, il kernel non saprebbe mai cosa ha fatto ogni thread (sebbene ad essi possano essere assegnate priorità diverse). In generale, comunque, gli scheduler di thread specifici per le applicazioni possono mettere a punto un'applicazione meglio di quanto possa fare il kernel.

2.6 Ricerca su processi e thread

Nel Capitolo 1, abbiamo dato uno sguardo alla ricerca odierna sulla struttura dei sistemi operativi. In questo e nei capitoli che seguono daremo un'occhiata ad argomenti di ricerca più circoscritti, cominciando dai processi. Come diverrà chiaro in breve tempo, alcuni argomenti sono più radicati che altri, e la maggior parte della ricerca tende a focalizzarsi su nuovi argomenti, piuttosto che su quelli a cui ha girato intorno per decenni.

Il concetto di processo è un esempio di argomento ben radicato: praticamente tutti i sistemi hanno una nozione di processo come contenitore per raggruppare risorse correlate come lo spazio di indirizzamento, i thread, i file aperti, le protezioni, i permessi, eccetera. Sistemi diversi effettuano il raggruppamento in maniera leggermente diversa, ma sono solo differenze di implementazione. L'idea base non è più discussa e c'è poca ricerca sull'argomento.

I thread sono un'idea più nuova che i processi, quindi c'è ancora qualche ricerca che procede su di essi. Hauser e altri (1993) hanno osservato come i veri programmi utilizzano nella pratica i thread, ed hanno trovato 10 paradigmi diversi per l'uso dei thread. La schedulazione dei thread (sia uniprocesso che multiprocessore) è ancora un argomento vicino e caro al cuore di alcuni ricercatori (Blumofe e Leiserson, 1994; Buchanan e Chien, 1997; Corbalan ed altri, 2000; Chandra ed altri, 2000; Duda e Cheriton, 1999; Ford e Susalla, 1996; Petrou ed altri, 1999). Tuttavia, pochi progettisti di sistema se ne vanno in giro torcendosi le mani, disperati per la mancanza di un adeguato algoritmo di schedulazione per i thread, quindi questo tipo di ricerca appare più spinta dai ricercatori che dalla domanda.

La sincronizzazione dei thread e la mutua esclusione sono strettamente correlati ai

thread. Negli anni '70 e '80, dall'argomento fu estratto tutto quello per cui valeva la pena, quindi attualmente non c'è molto lavoro su di esso e quello che c'è tende a essere focalizzato sulle prestazioni (per esempio Liedtke, 1993), su strumenti per trovare errori di sincronizzazione (Savage ed altri, 1997), oppure sulla modifica di concetti vecchi in nuovi modi (Tai e Carver, 1996; Trono, 2000). Infine, si stanno ancora producendo nuove librerie per i thread conformi a POSIX, su cui vengono fatte relazioni (Alfieri, 1994; Miller, 1999).

2.7 Sommario

Per mascherare l'effetto delle interruzioni, il sistema operativo mette a disposizione un modello concettuale fatto di processi sequenziali che lavorano in parallelo. I processi possono essere creati e terminati dinamicamente, ed ogni processo ha il proprio spazio di indirizzamento.

Per alcune applicazioni è utile avere più thread di controllo in un solo processo, che vengono schedulati indipendentemente; ognuno ha il proprio stack, ma tutti i thread in un processo condividono uno spazio di indirizzamento comune. I thread possono essere implementati nello spazio utente o nel kernel.

I processi possono comunicare fra loro usando le primitive di comunicazione fra processi, come semafori, monitor o messaggi, che vengono usate per garantire che due processi non possano mai essere contemporaneamente nelle loro sezioni critiche, una situazione che conduce al caos. Un processo può essere in esecuzione, pronto o bloccato, e può cambiare il proprio stato quando esso od un altro processo esegue una delle primitive di comunicazione. La comunicazione fra thread è simile.

Le primitive di comunicazione fra processi possono essere utilizzate per risolvere problemi quali produttore-consumatore, i filosofi a cena, i lettori-scrittori e il barbiere che dorme. Persino con queste primitive bisogna far attenzione per evitare errori e stalli.

Sono noti molti algoritmi di schedulazione, alcuni dei quali vengono usati principalmente nei sistemi batch, come lo shortest job first, altri sono comuni sia nei sistemi batch sia nei sistemi interattivi, come l'algoritmo round robin, la schedulazione con priorità, le code multilivello, la schedulazione garantita, la schedulazione a lotteria e la schedulazione a porzioni eque. Alcuni sistemi hanno una separazione netta tra meccanismi e politiche di schedulazione, che permettono agli utenti di controllare gli algoritmi di schedulazione.

PROBLEMI

1. In Figura 2.2 è mostrato lo stato di tre processi. In teoria, con tre stati, potrebbero verificarsi sei transizioni, due per uscire da ogni stato. Tuttavia, sono mostrate solo quattro transizioni. Esistono circostanze nelle quali si possono verificare una o entrambe le transizioni mancanti?
2. Supponete di progettare un'architettura avanzata per un calcolatore, che esegua lo scambio di processi via hardware, invece di avere le interruzioni. Di quali informazioni avrebbe bisogno la CPU? Descrivete come potrebbe funzionare lo scambio di processi via hardware.
3. Su tutti i calcolatori, almeno una parte dei gestori delle interruzioni è scritta in linguaggio assembler. Perché?

4. Quando un'interruzione o una chiamata di sistema trasferiscono il controllo al sistema operativo, di solito viene utilizzata un'area di stack del kernel separata da quella del processo interrotto. Perché?
5. Nel testo è stato affermato che il modello della Figura 2.6(a) non era adatto a un file server che usi una cache in memoria. Perché no? Ogni processo può avere la propria cache?
6. In Figura 2.7 l'insieme dei registri viene elencato rispetto ai thread invece che ai processi. Perché? Dopo tutto, la macchina ha solo un insieme di registri.
7. Se un processo multithread si biforca, si verifica un problema se il processo figlio ottiene le copie di tutti i thread del processo genitore. Supponete che uno dei thread originari fosse in attesa di dati in ingresso dalla tastiera: ora, due thread sono in attesa di dati in ingresso dalla tastiera, uno per ogni processo. Questo problema si verifica in processi a singolo thread?
8. Nella Figura 2.10 viene mostrato un server Web multithread. Se l'unico modo per leggere da un file è la normale chiamata di sistema bloccante `read`, pensate che, per implementare il server Web, vengano usati thread a livello utente o thread a livello kernel? Perché?
9. Perché un thread dovrebbe rilasciare la CPU chiamando `thread_yield`? Dopo tutto, dato che non c'è un'interruzione periodica di `clock`, potrebbe non riavere mai il controllo della CPU.
10. A un thread può essere tolta in anticipo la CPU (prerilascio) a causa di un'interruzione di `clock`? Se sì, in quali circostanze? Se no, perché?
11. In questo problema dovete confrontare la lettura di un file usando un file server a singolo thread e uno multithread. Occorrono 15 millisecondi per ottenere una richiesta di lavoro, smistarla e fare il resto della necessaria elaborazione, supponendo che i dati necessari siano in un blocco di cache. Se bisogna effettuare un'operazione su disco, come avviene in un terzo dei casi, sono richiesti 75 millisecondi aggiuntivi, durante i quali il thread è sospeso. Quante richieste al secondo può gestire il server se è a singolo thread? E se è multithread?
12. Nel testo, abbiamo descritto un server Web multithread, mostrando perché è migliore di un server a singolo thread e di un server implementato come una macchina a stati finiti. Esistono circostanze nelle quali un server a singolo thread potrebbe essere migliore? Date un esempio.
13. Nella discussione sulle variabili globali nei thread, abbiamo usato una procedura `create_global` per allocare memoria per un puntatore alla variabile piuttosto che per la variabile stessa. Questo è indispensabile o la procedura potrebbe lavorare altrettanto bene semplicemente con i valori stessi?
14. Considerate un sistema nel quale i thread vengono implementati completamente nello spazio utente, con il sistema a tempo di esecuzione che dà un'interruzione di `clock` una volta al secondo. Supponete che un'interruzione di `clock` si verifichi mentre qualche thread è in esecuzione nel sistema a tempo di esecuzione. Che problema potrebbe verificarsi? Potete suggerire un modo per risolverlo?
15. Supponete che un sistema operativo non abbia niente di simile alla chiamata di sistema `select` per vedere in anticipo se è sicuro leggere da un file, pipe o disponibile.

sitivo, ma che permetta di impostare sveglie per interrompere le chiamate di sistema bloccanti. È possibile implementare una libreria di thread nello spazio utente in queste condizioni? Dibattete.

16. Può il problema di inversione delle priorità discusso nella Sezione 2.3.4 verificarsi con thread di livello utente? Perché o perché no?
17. In un sistema con thread, c'è uno stack per ogni thread o uno stack per processo quando vengono utilizzati thread di livello utente? Cosa succede se i thread sono a livello kernel? Spiegare.
18. Cos'è una corsa critica?
19. Quando si progetta un calcolatore, di solito viene prima simulato da un programma che esegue un'istruzione per volta. Persino i multiprocessori vengono rigorosamente simulati in questo modo sequenziale. È possibile che si verifichi una corsa critica quando non ci sono eventi simultanei come questo?
20. La soluzione di attesa attiva che usa la variabile *turno* (Figura 2.20) funziona quando i due processi vengono eseguiti su un multiprocessore a memoria condivisa, cioè due CPU diverse che condividono una memoria comune?
21. La soluzione di Peterson per la mutua esclusione mostrata in Figura 2.21 funziona quando la schedulazione dei processi è con prerilascio? E quando è senza prerilascio?
22. Considerate una calcolatrice che non ha l'istruzione TSL ma ha un'istruzione per eseguire lo scambio tra il contenuto di un registro e di una parola di memoria in una singola azione indivisibile. Può essere usata per scrivere una routine *entra_nella_regione* come quella di Figura 2.22?
23. Dare una traccia di come un sistema operativo capace di disabilitare le interruzioni possa implementare i semafori.
24. Mostrate come i semafori con conteggio (cioè semafori che possono contenere un valore arbitrario) possono essere implementati utilizzando solo semafori binari e comuni istruzioni macchina.
25. Se un sistema ha solo due processi, ha senso usare una barriera per sincronizzarli? Perché o perché no?
26. Nella sezione 2.3.4, è stata descritta una situazione con un processo ad alta priorità, *H*, e un processo a bassa priorità, *L*, che porta *H* a ciclare all'infinito. Lo stesso problema si presenta anche se viene usata la schedulazione round robin invece che a priorità? Discutete.
27. Due thread nello stesso processo possono sincronizzarsi utilizzando un semaforo nel kernel, se i thread sono implementati dal kernel? E se sono implementati nello spazio utente? Supponete che nessun thread in nessun altro processo abbia accesso a quel semaforo. Discutete le vostre risposte.
28. La sincronizzazione tramite i monitor usa variabili condizione e due operazioni speciali, *wait* e *signal*. Una forma di sincronizzazione più generale dovrebbe avere una singola primitiva, *waituntil*, che abbia come parametro un predicato booleano arbitrario. Così, si potrebbe per esempio scrivere

`waituntil x < 0 or y + z < n`

La primitiva *signal* non sarebbe più necessaria. Questo schema è chiaramente più

- generale di quello di Hoare e Brinch Hansen, ma non viene usato. Perché? Suggerimento: si pensi all'implementazione.
29. Un ristorante fast-food ha quattro tipi di dipendenti: (1) i camerieri, che prendono le ordinazioni dei clienti; (2) i cuochi, che preparano il cibo; (3) gli impacchettatori, che mettono il cibo nelle borse e (4) i cassieri, che danno il cibo ai clienti e prendono il loro denaro. Ciascun dipendente può essere visto come un processo sequenziale che comunica. Quale forma di comunicazione fra processi usano? Collegate questo modello ai processi in UNIX.
 30. Supponete di avere un sistema a scambio di messaggi che usa le mailbox. Quando un processo manda un messaggio ad una mailbox piena o prova a ricevere messaggi da una mailbox vuota, non si blocca, ma ottiene un codice di errore. Il processo gestisce il codice di errore semplicemente provando e riprovando, finché l'operazione non ha successo. Questo schema porta a corse critiche?
 31. Nella soluzione del problema dei filosofi a cœua (Figura 2.33), perché la variabile di stato è impostata a *AFFAMATO* nella procedura *prendi_forchette*?
 32. Si consideri la procedura *posa_forchette* della Figura 2.33 e si supponga che la variabile *stato[i]* venga messa a *PENSANTE* dopo le due chiamate a *test*, anziché prima. Come influenzerebbe la soluzione questo cambiamento?
 33. Il problema dei lettori e scrittori può essere formulato in maniera diversa a seconda di quando si facciano partire le diverse classi di processi. Descrivete accuratamente tre varianti diverse del problema, ciascuna delle quali favorisca (o non favorisca) una qualche categoria di processi. Per ciascuna variazione, si specifichi cosa succede quando un lettore o uno scrittore diventa pronto ad accedere alla base di dati e cosa succede quando un processo ha finito di utilizzarla.
 34. Il calcolatore CDC 6600 poteva gestire fino a 10 processi di ingresso/uscita contemporaneamente grazie ad una interessante forma di round robin, chiamata **processor sharing** (condivisione di processore). Dopo l'esecuzione di ogni istruzione, si verificava un cambio di contesto. Così l'istruzione 1 proveniva dal processo 1, la 2 dal processo 2 e così via. Il cambio di contesto era implementato tramite hardware specializzato e l'overhead era nullo. Se un processo richiedeva T secondi per il completamento in assenza di competizione, quanto tempo richiedeva se veniva usato il processor sharing con n processi?
 35. Gli scheduleri round robin mantengono normalmente una lista di tutti i processi pronti, nella quale ciascun processo compare una sola volta. Cosa potrebbe accadere se un processo apparisse due volte nella lista? Riuscite ad immaginare un motivo per cui questo potrebbe essere permesso?
 36. Si può misurare se probabilmente un processo sarà CPU bound o I/O bound tramite l'analisi del codice sorgente? Come si può determinarlo a tempo di esecuzione?
 37. Nella sezione "Quando schedulare" è stato accennato che, qualche volta, la schedulazione potrebbe essere migliorata se un processo importante, quando si blocca, giocasse un ruolo nella selezione del prossimo processo da eseguire. Descrivete una situazione in cui questo potrebbe essere usato e spiegate come.
 38. Misure eseguite su un certo sistema hanno mostrato che il processo medio viene eseguito per un tempo pari a T prima di bloccarsi per un'operazione di ingresso/uscita. Un cambio di contesto richiede un tempo S , che viene di fatto sprecato (overhead). Per uno scheduler round robin con quanto Q , dare la formula per

l'efficienza della CPU per ciascuno dei seguenti casi:

- (a) $Q = \infty$
 - (b) $Q > T$
 - (c) $S < Q < T$
 - (d) $Q = S$
 - (e) Q vicino allo 0
39. Cinque job stanno aspettando di andare in esecuzione. La stima dei loro tempi di esecuzione è di 9, 6, 3, 5 e X. In che ordine dovrebbero essere mandati in esecuzione per minimizzare il tempo di risposta medio? (La vostra risposta dipenderà da X.)
40. Cinque job batch, da A a E, arrivano al centro di calcolo circa nello stesso istante. Hanno un tempo di esecuzione stimato di 10, 6, 2, 4 e 8 minuti, rispettivamente. Le loro priorità (determinate esternamente) sono rispettivamente 3, 5, 2, 1 e 4, dove 5 rappresenta la massima priorità. Per ognuno dei seguenti algoritmi di schedulazione, determinare il tempo medio di turnaround. Si ignori l'overhead dovuto al cambio di contesto.
- (a) Round robin.
 - (b) Schedulazione a priorità.
 - (c) Primo-arrivato, primo-servito (mandati in esecuzione nell'ordine 10, 6, 2, 4, 8).
 - (d) Shortest job first.
- Nel caso (a) si supponga che il sistema sia multiprogrammato e che ogni job ottenga la sua ragionevole quota di CPU. Nei casi da (b) a (d) si supponga che solo un job alla volta venga mandato in esecuzione e vi rimanga finché finisce. Tutti i job sono CPU bound.
41. Un processo che è in esecuzione su un CTSS ha bisogno di 30 quanti per completare la sua esecuzione. Quante volte dovrà essere scambiato, compresa la prima (prima che venga effettivamente eseguito)?
42. Siete in grado di pensare a un modo per evitare che il sistema a priorità CTSS venga imbrogliato dal premere a caso il tasto INVIO?
43. Si vuole usare l'algoritmo di invecchiamento con $\alpha = 1/2$ per prevedere i tempi di esecuzione. Le quattro esecuzioni precedenti, dalla prima alla più recente, sono di 40, 20, 40 e 15 millisecondi. Qual è il tempo previsto per la prossima esecuzione?
44. Un sistema soft real time ha quattro eventi periodici con periodi di 50, 100, 200 e 250 millisecondi ciascuno. Supponete che i quattro eventi richiedano rispettivamente 35, 20, 10 e x millisecondi di tempo di CPU. Qual è il più grosso valore di x per il quale il sistema è schedulabile?
45. Spiegare perché comunemente viene usato uno scheduler a due livelli.
46. Considerate un sistema nel quale si desidera separare politiche e meccanismi per la schedulazione dei thread del kernel. Proponete un modo per raggiungere questo scopo.
47. Scrivete uno script di shell che produce un file di numeri sequenziali leggendo l'ultimo numero nel file, aggiungendogli 1 e aggiungendo il risultato al file. Eseguite una volta lo script in background e una in foreground, accedendo entrambe le volte allo stesso file. Quanto ci vuole prima che una corsa critica si manifesti? Qual è la regione critica? Modificate lo script per evitare la corsa. Suggerimento: utilizzate

ln file file.lock

per bloccare il file di dati.

48. Supponete di avere un sistema operativo che fornisce i semafori. Implementate un sistema di messaggi e scrivete le procedure per inviare e ricevere messaggi.
49. Risolvere il problema dei filosofi a cena con i monitor anziché con i semafori.
50. Supponete che un'università voglia ostentare quanto è politicamente corretta applicando la dottrina della Corte Suprema degli Stati Uniti "Separato ma uguale è intrinsecamente disuguale" ai sessi come alle razze, mettendo fine alla pratica di vecchia data dei bagni separati in base ai sessi nell'Università. Comunque, come concessione alla tradizione, decreta che quando una donna è in un bagno, un'altra donna può entrare, ma nessun uomo, e viceversa. Un segno con un pennarello cancellabile sulla porta di ciascun bagno indica in quale dei tre possibili stati ci si trova:

- Vuoto
- C'è una donna
- C'è un uomo

Nel vostro linguaggio di programmazione preferito, scrivete le procedure seguenti: *vuole_entrare_una_donna*, *vuole_entrare_un_uomo*, *esce_una_donna*, *esce_un_uomo*. Potete usare le tecniche di conteggio e sincronizzazione che preferite.

51. Riscrivete il programma di Figura 2.20 per gestire più di due processi.
52. Scrivete un problema produttore-consumatore che usi i thread e condivida un buffer comune. Comunque, non usate i semafori o altre primitive di sincronizzazione per salvaguardare le strutture dati condivise, semplicemente lasciate che ogni thread acceda ad esse quando vuole. Usate *sleep* e *wakeup* per gestire le condizioni di vuoto e pieno. Vedete quanto ci vuole prima che si verifichi una corsa critica fatale. Ad esempio, potreste avere il produttore che, ogni tanto, stampa un numero. Non stampate più di un numero al minuto perché le operazioni di ingresso/uscita possono influenzare le corse critiche.
53. Un processo può esser messo in una coda round robin più di una volta per dargli una priorità più alta. Eseguire più volte un programma, che ogni volta lavora su una parte diversa di un parco dati, può avere gli stessi effetti. Scrivete prima un programma, dato un elenco di numeri, verifica se sono numeri primi, poi escogitate un metodo per permettere a istanze multiple del programma di essere eseguite contemporaneamente in modo che nessuna coppia di istanze del programma lavori sullo stesso numero. In realtà, potete scorrere la lista più velocemente che eseguendo copie multiple del programma? Notate che i vostri risultati dipenderanno da quali altre cose sta facendo il vostro calcolatore: su un personal computer che esegue solo istanze del programma non vi dovrete aspettare un miglioramento, mentre su un sistema con altri processi, in questo modo dovreste essere in grado di ottenere una porzione maggiore della CPU.

DEADLOCK

I sistemi di calcolo hanno molte risorse che possono essere usate solo da un processo alla volta, come le stampanti, i dispositivi a nastro e gli spazi nelle tabelle interne del sistema. Una situazione di confusione si ha nel caso in cui due processi scrivono simultaneamente sulla stampante; allo stesso modo si avrà un file system inconsistente se due processi usano lo stesso spazio nelle tabelle del file system. Perciò tutti i sistemi operativi possono assegnare temporaneamente l'accesso esclusivo di un processo ad una certa risorsa.

In molte applicazioni un processo ha bisogno di accedere in modo esclusivo non ad una sola risorsa bensì a molte. Supponiamo, ad esempio, che due processi vogliano registrare su CD un documento digitalizzato da uno scanner: il processo A richiede il permesso di utilizzare lo scanner ed è autorizzato; il processo B è programmato in maniera differente e richiede dapprima il masterizzatore CD e viene autorizzato. A questo punto A richiede l'uso del masterizzatore CD, ma la richiesta viene rifiutata finché B non lo rilascia; sfortunatamente, invece di rilasciare il masterizzatore CD, B richiede lo scanner. A questo punto entrambi i processi vengono sospesi, e, viste le premesse, rimarranno sospesi per un tempo infinito: una situazione di questo tipo è detta **deadlock** o **stallo**.

I deadlock possono apparire in molte situazioni; ad esempio, molti uffici hanno una LAN con molti computer collegati ad essa: spesso i dispositivi come scanner, CD, stampanti e registratori a nastro sono collegati alla rete come risorse condivise, disponibili ad ogni utente e da ogni macchina. Se questi dispositivi possono essere riservati in remoto (ad esempio, dalla macchina di casa di un utente), si può verificare lo stesso tipo di stallo descritto in precedenza; inoltre, situazioni più complicate possono causare stallo, con tre, quattro o più dispositivi ed utenti.

I deadlock possono apparire in molte situazioni, oltre che con i dispositivi di input/output dedicati. Ad esempio, in un sistema che gestisce basi di dati, un programma può bloccare (lock) diversi record per evitare le corse critiche. Nel caso in cui un processo *A* blocchi il record *R1*, un processo *B* blocchi il record *R2* e poi entrambi cerchino di bloccare il record già bloccato dall'altro, si ha ancora una condizione di stallo. Praticamente, si possono verificare condizioni di stallo sia su risorse hardware sia su risorse software.

In questo capitolo si considereranno i deadlock più da vicino, si vedrà come si presentano e si studierà come prevenirli o evitarli. Sebbene questo materiale riguardi i deadlock nel contesto dei sistemi operativi, questi si presentano anche nei sistemi di basi di dati e in molti altri contesti informatici, quindi questo materiale è in realtà applicabile ad un'ampia varietà di sistemi multiprocessori. Su questo argomento sono state scritte moltissimi lavori: due bibliografie sullo stallo si trovano nella rivista *Operating Systems Review* e possono senz'altro essere usate come riferimento (Newton 1979; Zobel 1983). Sebbene queste bibliografie siano datate, la maggior parte del lavoro svolto sui deadlock è stato realizzato prima del 1980, perciò queste risultano ancora utili.

3.1 Risorse

I deadlock possono verificarsi quando ad un processo viene garantito l'accesso esclusivo a dispositivi, file, eccetera. Per rendere la trattazione dei deadlock più generale possibile, si farà riferimento sempre agli oggetti a cui si vuole accedere in maniera esclusiva con il termine **risorse**. Una risorsa può essere un dispositivo hardware (ad esempio un lettore di nastro), oppure alcune informazioni (ad esempio un record bloccato in una base di dati). In generale, un computer può avere molte risorse diverse, ed alcune di queste possono essere presenti in più di un esemplare, ad esempio si possono avere tre lettori di nastro. Quando più risorse dello stesso tipo sono presenti nel sistema, si può usarne una qualsiasi per soddisfare una richiesta di quel tipo di risorse. Riassumendo, una risorsa è qualsiasi cosa che possa essere usata da un unico processo in un certo istante di tempo.

3.1.1 Risorse prerilasciabili e non prerilasciabili

Vi sono due tipi di risorse: prerilasciabili e non prerilasciabili. Una **risorsa prerilasciabile** (*preemptable resource*) è una risorsa che si può togliere al processo che la sta usando senza provocare effetti dannosi. Un esempio è la memoria centrale: si consideri, come esempio, un sistema con 32 MB di memoria utente, una stampante e due processi di 32 MB che intendono stampare qualcosa. Il processo *A* richiede ed ottiene la stampante, iniziando il calcolo dei valori da stampare: prima che abbia terminato il calcolo supera il quanto di tempo ad esso assegnato ed è sospeso.

Ora è in esecuzione il processo *B* e prova, senza successo, ad ottenere la stampante; potenzialmente si ha una situazione di stallo, infatti *A* ha la stampante e *B* la memoria, e nessuno dei due processi può avanzare senza la risorsa occupata dall'altro. Per fortuna, *B* può rilasciare la memoria cedendola ad *A*, che può continuare l'esecuzione, fare le sue stampe e quindi rilasciare la stampante, per cui non ci sono deadlock.

Al contrario, una **risorsa non prerilasciabile** (*non preemptable resource*) non può essere ceduta ad un altro processo senza provocare il fallimento dell'esecuzione in atto. Se un processo ha iniziato a scrivere un CD e lo si priva improvvisamente del masterizzatore CD, assegnandolo ad un altro processo, si avrà un CD incomprensibile: i masterizzatori CD non sono risorse prerilasciabili.

In generale, i deadlock si hanno con le risorse non prerilasciabili; gli eventuali deadlock che coinvolgono risorse prerilasciabili di solito si possono risolvere riallocando le risorse tra i processi. Per questo motivo si porrà maggior attenzione alle risorse non prerilasciabili.

Per usare una risorsa, è necessaria la seguente sequenza di eventi:

1. Richiedere la risorsa.
2. Usare la risorsa.
3. Rilasciare la risorsa.

Se la risorsa non è disponibile al momento in cui viene richiesta, il processo che ha prodotto la richiesta viene messo in stato di attesa: in alcuni sistemi operativi, il processo viene automaticamente bloccato e fatto ripartire quando la risorsa diventa disponibile; in altri sistemi, invece, la richiesta può fallire e restituire un codice di errore, lasciando così al processo chiamante il compito di aspettare un po' di tempo per poi riprovare ad acquisire la risorsa.

Un processo, a cui la risorsa è stata appena negata, di solito andrà in un ciclo di attesa, ed in un secondo momento proverà a richiederla. Sebbene questo processo non sia bloccato in tutte le sue attività, esso risulta esserlo di fatto poiché non può lavorare in maniera utile; da questo momento in avanti si supporrà che quando una risorsa viene negata ad un processo, questo sarà messo in attesa.

Che cosa si debba fare per richiedere una risorsa, dipende in gran parte dal sistema in cui ci troviamo. In alcuni sistemi troviamo una chiamata di sistema **request** che permette ad un processo di richiedere le risorse; in altri, le uniche risorse rese disponibili dal sistema operativo sono i file speciali, che possono essere aperti da un solo processo alla volta, tramite la solita chiamata **open**. Se il file è già in uso, il chiamante viene bloccato finché il processo che lo ha in uso non lo chiude.

3.1.2 Acquisizione di risorse

Per alcuni tipi di risorse, come i record nei sistemi di basi di dati, è compito dei processi utenti gestire l'utilizzo delle risorse: un modo possibile è quello di associare a cias-

```
typedef int semaforo;
semaforo risorsa_1;

void processo_A(void) {
    down(&risorsa_1);
    usa_risorsa_1();
    up(&risorsa_1);
}
```

(a)

```
typedef int semaforo;
semaforo risorsa_1;
semaforo risorsa_2;

void processo_A(void) {
    down(&risorsa_1);
    down(&risorsa_2);
    usa_entrambe_risorse();
    up(&risorsa_2);
    up(&risorsa_1);
}
```

(b)

Figura 3.1 Utilizzo dei semafori per proteggere le risorse. [a] Una risorsa. [b] Due risorse.

```

typedef int semaforo;
semaforo risorsa_1;
semaforo risorsa_2;

void processo_A(void) {
    down(&risorsa_1);
    down(&risorsa_2);
    usa_entrambe_risorse();
    up(&risorsa_2);
    up(&risorsa_1);
}

void processo_B(void) {
    down(&risorsa_1);
    down(&risorsa_2);
    usa_entrambe_risorse();
    up(&risorsa_2);
    up(&risorsa_1);
}

```

(a)

```

semaforo risorsa_1;
semaforo risorsa_2;

void processo_A(void) {
    down(&risorsa_1);
    down(&risorsa_2);
    usa_entrambe_risorse();
    up(&risorsa_2);
    up(&risorsa_1);

}

void processo_B(void) {
    down(&risorsa_2);
    down(&risorsa_1);
    usa_entrambe_risorse();
    up(&risorsa_1);
    up(&risorsa_2);
}

```

(b)

Figura 3.2 [a] Codice privo di stallo. [b] Codice con stallo potenziale.

scuna di queste un semaforo, inizializzandoli tutti a 1; allo stesso modo possono essere utilizzati i mutex. I tre passi elencati in precedenza, vengono implementati come una down sul semaforo, per poter acquisire la risorsa, utilizzarla, ed infine una up sulla risorsa per rilasciarla. Questi passi sono mostrati in Figura 3.1 (a).

Qualche volta i processi necessitano di due o più risorse e quindi le acquisiscono in maniera sequenziale, come mostrato in Figura 3.1(b); e se si ha bisogno di più di due risorse, vengono acquisite una dopo l'altra.

Perciò, finché è coinvolto un solo processo, tutto funziona alla perfezione: naturalmente, con un solo processo, non c'è bisogno di acquisire formalmente una risorsa, poiché non c'è competizione per esse.

Consideriamo ora la situazione con due processi, *A* e *B*, e due risorse. Due scenari vengono mostrati in Figura 3.2: in Figura 3.2(a), entrambi i processi richiedono le risorse nello stesso ordine; in Figura 3.2(b), le richiedono con ordine differente; questa differenza può sembrare minima, ma non lo è.

In Figura 3.2(a), uno dei processi acquisirà la prima risorsa prima dell'altro; tale processo, successivamente acquisirà con successo anche la seconda risorsa ed eseguirà il suo compito: se l'altro processo cerca di acquisire la risorsa 1 prima che sia rilasciata, verrà bloccato, finché la risorsa diverrà disponibile.

In Figura 3.2(b) la situazione è differente: potrebbe accadere che uno dei due processi acquisisca entrambe le risorse e blocchi effettivamente l'altro processo, finché non abbia terminato; tuttavia potrebbe anche accadere che il processo *A* acquisisca la risorsa 1 ed il processo *B* la risorsa 2: entrambi saranno quindi bloccati quando tenteranno di acquisire l'altra risorsa, nessun processo entrerà mai più in esecuzione e dunque questa situazione è un deadlock!

Qui si vede come ciò che può apparire come una minima differenza nello stile del codice - quale risorsa acquisire prima - possa fare la differenza tra un programma che funziona ed un programma che fallisce in un modo difficile da rilevare. E siccome i deadlock possono comparire così facilmente, sono state compiute molte ricerche sul modo in cui trattarli. Questo capitolo discute i deadlock in dettaglio e cosa si può fare con essi.

3.2 Introduzione ai deadlock

Il deadlock può essere formalmente definito come segue:

Un insieme di processi si trova in una situazione di stallo se ogni processo dell'insieme aspetta un evento che solo un altro processo dell'insieme può provocare.

Poiché tutti i processi stanno aspettando, nessuno di essi potrà mai causare l'evento che potrebbe far ripartire un altro processo, e così tutti continueranno ad aspettare all'infinito. In questo modello supponiamo che i processi abbiano solo un thread, e che non vi siano possibili interruzioni che risveglieranno un processo bloccato. La condizione di assenza di interruzioni è necessaria per evitare che un altro processo in stallo sia risvegliato da, diciamo, un allarme, e questo provochi eventi che rilascino altri processi nell'insieme.

Nella maggior parte dei casi, l'evento che un processo aspetta è proprio il rilascio di una risorsa da parte di un altro processo appartenente all'insieme. In altre parole, ogni membro di un insieme di processi in stallo, aspetta una risorsa che può essere rilasciata solo da un altro processo che si trova in stallo. In questo caso, nessun processo può passare allo stato di esecuzione, nessuno può rilasciare una risorsa e nessun processo può essere sbloccato. Il numero dei processi e il genere delle risorse e delle richieste non è influente per questa trattazione: questo risultato vale per ogni tipo di risorsa, sia hardware che software.

3.2.1 Condizioni per il deadlock

Coffman e altri (1971) hanno dimostrato che devono valere quattro condizioni perché possa verificarsi una situazione di stallo:

1. Condizione di mutua esclusione: ogni risorsa è assegnata ad un solo processo, oppure è disponibile.
2. Condizione di prendi e aspetta (hold and wait): i processi che già hanno richiesto ed ottenuto delle risorse ne possono richiedere delle altre.
3. Condizione di mancanza di prerilascio: le risorse che sono già state assegnate a un processo non gli possono essere tolte in modo forzato, ma devono essere rilasciate volontariamente dal processo che le detiene.
4. Condizione di attesa circolare: deve esistere una catena circolare di processi, ognuno dei quali aspetta il rilascio di una risorsa da parte del processo che lo segue.

Queste quattro condizioni devono essere tutte presenti perché si abbia un deadlock, mentre non può verificarsi se ne manca anche solo una.

Vale la pena notare che ogni condizione è legata ad una politica che il sistema può avere o meno: si può assegnare una data risorsa a più di un processo alla volta? Un proces-

so può acquisire una risorsa e chiederne un'altra? Si possono prerilasciare le risorse? Possono esistere le attese circolari? Nel seguito vedremo come poter attaccare i deadlock, cercando di negare alcune di questi condizioni.

3.2.2 Modelli per le situazioni di stallo

Holt (1972) ha dimostrato come queste quattro condizioni possano essere modellate usando grafi orientati. I grafi hanno due tipi di nodi: i processi, rappresentati dai cerchietti, e le risorse, rappresentate dai quadratini. La presenza di un arco da un nodo di tipo risorsa (quadrato) a un nodo di tipo processo (cerchio) significa che la risorsa è stata prima richiesta dal processo, poi ottenuta, ed attualmente è posseduta dal processo stesso. Nella Figura 3.3(a), la risorsa R è assegnata al processo A.

Un arco da un processo a una risorsa indica che il processo è attualmente bloccato in attesa della risorsa. Nella Figura 3.3(b) il processo B è in attesa della risorsa S. Nella Figura 3.3(c) si può osservare un deadlock: il processo C sta aspettando la risorsa T, la quale è attualmente in uso da parte del processo D. Il processo D non rilascia la risorsa poiché sta aspettando la risorsa U, attualmente in uso da parte di C; pertanto entrambi i processi sono in una situazione di attesa infinita. Nel grafo, la presenza di un ciclo indica un deadlock che coinvolge i processi e le risorse presenti nel ciclo; in questo esempio, il ciclo è C-T-D-U-C.

Si consideri ora un esempio di come possano essere usati i grafi delle risorse. Si immagini di avere tre processi, A, B e C, e tre risorse, R, S e T; le richieste e i rilasci di questi tre processi sono mostrati in Figura 3.4(a), (b) e (c). Il sistema operativo è libero, in ogni istante, di mandare in esecuzione un qualunque processo non bloccato, cosicché può decidere di mandare in esecuzione A finché non avrà finito, poi completare B e infine C.

Quest'ordine di esecuzione non porta a situazioni di stallo (perché non c'è competizione per l'uso delle risorse), ma non presenta alcuna forma di parallelismo: oltre a richiedere e a rilasciare risorse, i processi calcolano e fanno dell'input/output, e quando i processi vengono eseguiti in maniera sequenziale, non è possibile sfruttare i tempi di attesa dell'input/output di un processo per eseguirne un altro. Questo dimostra come l'esecuzione di un processo in maniera strettamente sequenziale non sia in generale una buona soluzione. D'altra parte, se nessun processo fa dell'input/output, la politica di eseguire prima i processi più piccoli è migliore di quella del round robin cosicché, sotto certe ipotesi, eseguire i processi in maniera sequenziale può risultare la migliore soluzione.

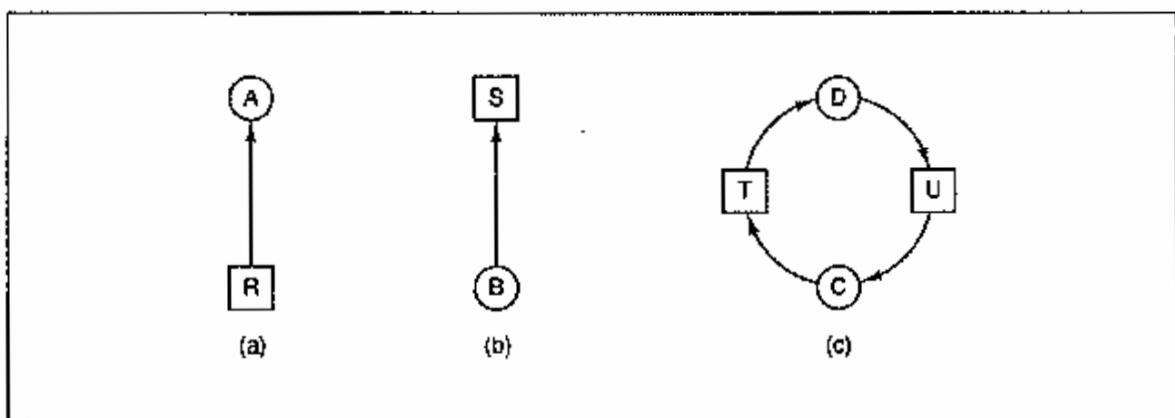


Figura 3.3 Grafi di allocazione delle risorse. [a] Acquisizione di una risorsa. [b] Richiesta di una risorsa. [c] Stallo.

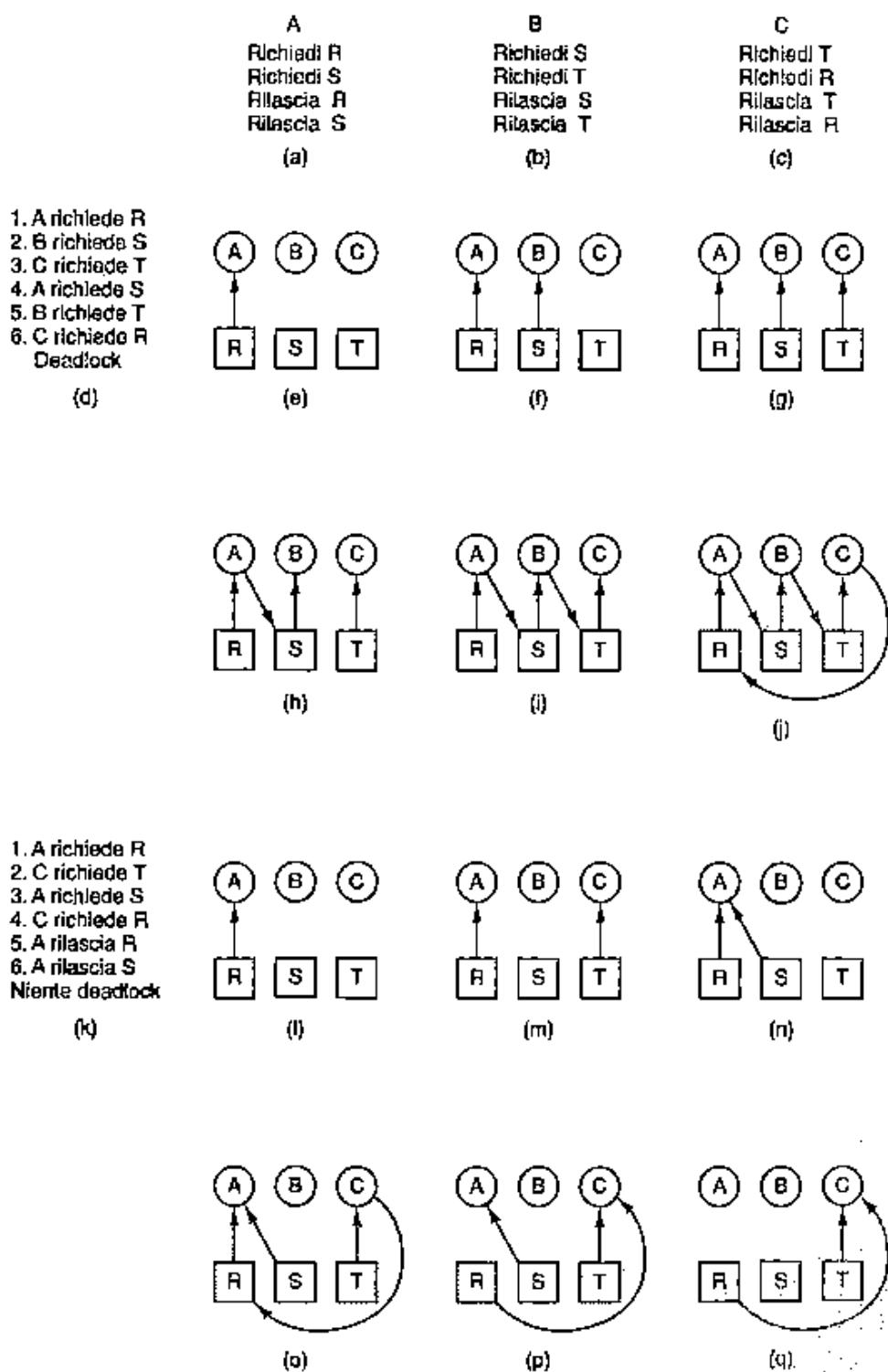


Figura 3.4 Un esempio di come può avvenire il deadlock e come lo si possa evitare.

Si supponga, ora, che i processi siano composti da fasi di calcolo e da fasi di I/O, in maniera tale da rendere ragionevole un algoritmo di scheduling di tipo round robin. La richiesta delle risorse può presentarsi nell'ordine presentato in Figura 3.4(d); in tal caso, i sei grafi delle risorse risultanti sono quelli di Figura 3.4(e)-(j). Dopo che è stata fatta la richiesta 4, A si blocca aspettando S, come da Figura 3.4(h); nei due passi immediatamente successivi, si bloccano B e C producendo un ciclo che corrisponde al deadlock di Figura 3.4(j).

Ad ogni modo, come si è già osservato, il sistema operativo non è assolutamente forzato a mandare in esecuzione i processi in un ordine speciale; in particolare, se assegnare una certa risorsa può portare ad una situazione di stallo, il sistema operativo può semplicemente sospendere il processo richiedente senza assegnarla (cioè può semplicemente evitare di schedularlo), finché l'assegnazione non possa avvenire senza stallo. Nel caso della Figura 3.4, se il sistema operativo si rendesse conto dell'imminente stallo, potrebbe sospendere B anziché assegnargli la risorsa S; eseguendo solo A e C, si otterrebbero le richieste e i rilasci mostrati in Figura 3.4(k) anziché quelli di Figura 3.4(d); tale sequenza dà luogo al grafo di Figura 3.4(l)-(q) che non presenta situazioni di stallo.

Dopo il passo (q), si può assegnare al processo B la risorsa S poiché A ha già finito e C ha già tutto ciò che gli serve per essere eseguito. Anche se B si bloccasse sulla richiesta di T, non si avrebbe stallo poiché B dovrebbe semplicemente aspettare che C abbia finito.

Più avanti, in questo stesso capitolo, si studierà un algoritmo ben preciso per prendere decisioni relative all'allocazione delle risorse in maniera tale che non si verifichino situazioni di stallo. Il punto che deve essere chiaro fin da ora, è che i grafi delle risorse sono uno strumento che permette di verificare se una certa sequenza di richieste e rilasci di risorse porta o meno ad una situazione di stallo. L'unica cosa che si deve fare è di seguire le richieste ed i rilasci passo per passo e, ogni volta, controllare se il grafo contiene cicli: se non li contiene non siamo in presenza di situazioni di stallo. Anche se la nostra discussione circa i grafi delle risorse ha esaminato il caso di una singola risorsa di ogni tipo, i grafi delle risorse possono essere generalizzati in modo da trattare anche il caso in cui si abbia più di una risorsa della stessa tipo (Holt 1972).

In generale, per trattare adeguatamente le situazioni di stallo si possono impiegare quattro strategie:

1. Ignorare semplicemente il problema: forse se tu lo ignori, questo ti ignorerà.
2. Rilevare e risolvere: quando si verifica una situazione di stallo, occorre scoprirla e risolverla.
3. Cercare dinamicamente di evitare le situazioni di stallo con un'accorta politica di allocazione delle risorse.
4. Prevenire le situazioni di stallo, negando una delle quattro condizioni necessarie.

Nei prossimi quattro paragrafi esamineremo nel dettaglio queste strategie.

3.3 L'algoritmo dello struzzo

L'approccio più semplice è rappresentato dall'algoritmo dello struzzo: nascondete la vostra testa nella sabbia e pretendete che il problema non esista. In effetti, è idiota citare gli struzzi come esempio di animali passivi: essi possono correre a 60 Km/ora e sferrare calci così potenti da uccidere un leone che li considerasse grossi polli adatti per un pranzetto.

Personne differenti reagiscono diversamente ad un simile approccio: i matematici lo

trovano totalmente inaccettabile, e sostengono che le situazioni di stallo devono essere evitate ad ogni costo; gli ingegneri si chiedono quanto spesso si presentino situazioni di stallo, quanto spesso il sistema cada per altre ragioni e quanto sia seria una situazione di stallo. Se le situazioni di stallo si verificano mediamente una volta ogni cinque anni ma il sistema cade una volta alla settimana per problemi di hardware, errori dei compilatori e del sistema operativo, la maggior parte degli ingegneri non sono disposti a pagare alcunché in termini di prestazioni o di comodità d'uso per l'eliminazione dei deadlock.

A rendere ancora più specifico questo contrasto, nella maggior parte dei sistemi operativi esistono delle situazioni potenziali di stallo di cui non ci si accorge nemmeno, e tanto meno le si risolve automaticamente. Il numero totale dei processi presenti nel sistema è determinato dal numero di elementi della tabella dei processi; così le posizioni della tabella dei processi sono una risorsa limitata. Se una fork fallisce perché la tabella è piena, un approccio ragionevole per il programma può essere quello di aspettare per un intervallo di tempo di lunghezza casuale, e poi di provare di nuovo ad eseguire la fork.

Ora, si supponga che il sistema UNIX abbia 100 posizioni nella tabella dei processi, e si supponga che siano in esecuzione 10 programmi ognuno dei quali deve creare 12 (sotto)processi. Dopo che ogni processo ha creato 9 nuovi processi, i 10 processi originali ed i 90 appena creati hanno esaurito tutte le posizioni della tabella. Ognuno dei processi originali si trova quindi in una situazione di ciclo infinito in cui cerca di eseguire la fork, fallisce e quindi ci riprova: è una situazione di stallo. La probabilità che questo accada è minima, ma comunque esiste. Dobbiamo quindi abbandonare i processi e le fork per eliminare il problema?

Analogamente, il numero dei file che possono essere aperti contemporaneamente è limitato dalle dimensioni della tabella degli i-node, cosicché ci si trova di fronte ad un problema del tutto simile al precedente, quando questa tabella viene riempita. Lo spazio sul disco usato per le operazioni di swapping è un'altra risorsa limitata; in effetti, ogni singola tabella che si trovi nel sistema operativo rappresenta una risorsa limitata. Si deve abolire tutto ciò perché può accadere che, in un insieme di n processi, ognuno di essi richieda $1/n$ risorse e poi ciascuno ne richieda una parte ulteriore?

L'approccio adottato dalla maggior parte dei sistemi operativi, tra cui UNIX e Windows, è semplicemente quello di ignorare il problema, e si basa sull'ipotesi che la maggior parte degli utenti preferisce una situazione occasionale di stallo ad una regola che imponga ad ogni utente di usare un unico processo, di aprire un unico file o di usare un'unica copia di una qualsiasi altra risorsa. Se le situazioni di stallo potessero essere eliminate senza sforzo, non ci sarebbe alcuna discussione; il problema, come si vedrà fra poco, è che il prezzo di una simile eliminazione è alto, in termini di limitazioni imposte sui processi. Così si è posti di fronte al dilemma della scelta fra correttezza e convenienza, e ci si trova nelle condizioni di dover discutere quale delle due sia la più importante per le diverse categorie di utenti. In questa ipotesi, non si possono trovare soluzioni generali.

3.4 Identificazione e risoluzione dei deadlock

Una seconda tecnica è l'identificazione e la risoluzione: il sistema non tenta di impedire il verificarsi dei deadlock, ma, al contrario, permette che si abbia il deadlock, prova a scoprire quando ciò accade, e poi prova a risolvere la situazione di stallo dopo che si è verificata. In questa sezione esamineremo alcuni modi con cui si possono scoprire i deadlock, ed alcuni modi con cui si possono risolvere le situazioni di stallo.

3.4.1 Identificazione del deadlock con una risorsa per ogni classe

Per iniziare si guarda il caso più semplice: esiste una sola risorsa di ogni tipo. Un sistema di questo tipo potrebbe avere uno scanner, un masterizzatore CD, un plotter ed un dispositivo a nastro, ma non più di uno per ogni classe di risorsa. In altre parole, per il momento si stanno escludendo sistemi con due stampanti; di questi si parlerà più tardi usando un metodo diverso.

Per un sistema di questo tipo si costruirà un grafo di risorse del tipo illustrato in Figura 3.3; se questo grafo contiene uno o più cicli allora esiste un deadlock, ed ogni processo che fa parte di un ciclo è in stallo; se non esistono cicli, il sistema non è in stallo.

Come esempio di un sistema più complesso di quello che si è visto si consideri un sistema con 7 processi, da A a G, e 6 risorse, da R a W. Il possesso delle risorse e le richieste sono le seguenti:

1. Il processo A occupa R e richiede S.
2. Il processo B non occupa risorse ma richiede T.
3. Il processo C non occupa risorse ma richiede S.
4. Il processo D occupa U e richiede sia S sia T.
5. Il processo E occupa T e richiede V.
6. Il processo F occupa W e richiede S.
7. Il processo G occupa V e richiede U.

La domanda che ci si pone è: "Questo sistema è in stallo, e se così fosse, quali sono i processi coinvolti?"

Per rispondere a questa domanda si può costruire il grafo delle risorse di Figura 3.5(a), che, come si vede a colpo d'occhio, contiene un ciclo. Il ciclo è mostrato in Figura 3.5(b), in cui si vede che i processi D, E e G sono tutti in stallo. I processi A, C e F non sono in stallo perché S può essere occupata da ognuno di essi che, terminando, la renderà libera facendo sì che gli altri due a turno possano occuparla e terminare.

Sebbene sia relativamente semplice trovare i processi in stallo guardando semplicemente il grafo, nei sistemi ci vuole un algoritmo formale per scoprire i deadlock. Si conoscono molti algoritmi per la scoperta di cicli nei grafi orientati; ecco un semplice algorit-

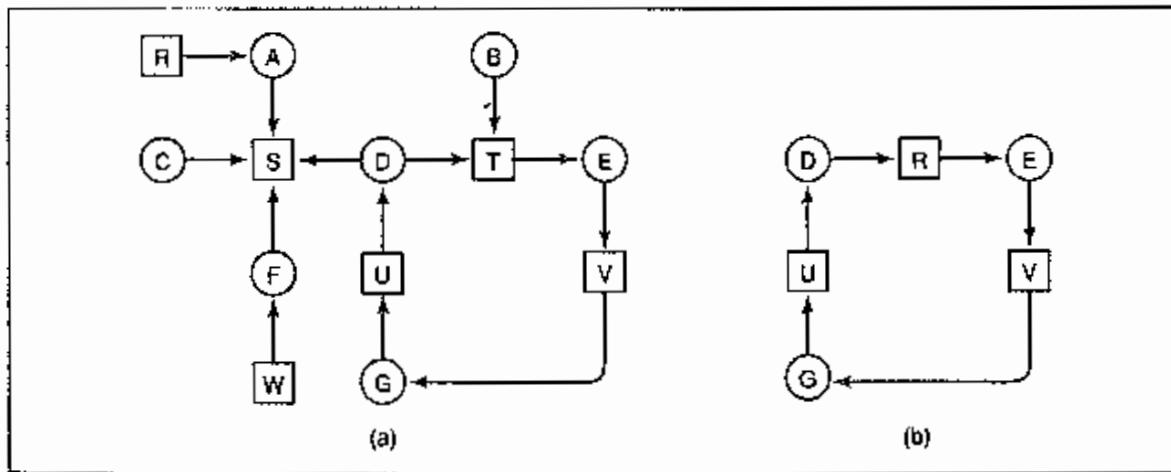


Figura 3.5 (a) Un grafo delle risorse. (b) Un ciclo estratto da (a).

mo che ispeziona un grafo e termina quando ha trovato un ciclo oppure quando ha appurato che non ne esiste nessuno. Questo algoritmo usa come struttura dati una lista di nodi L ; durante l'esecuzione dell'algoritmo gli archi saranno marcati per indicare che sono già stati ispezionati. L'algoritmo agisce con i passi seguenti:

1. Per ogni nodo N del grafo, eseguire i 5 passi successivi con N come nodo di partenza.
2. Inizializzare L come lista vuota e definire tutti gli archi come non marcato.
3. Aggiungere il nodo corrente alla fine di L e controllare che il nodo non appaia in L due volte. Se così fosse il grafo contiene un ciclo (elencato in L) e l'algoritmo termina.
4. Dal nodo dato, guardare se vi sono archi uscenti non marcato, in questo caso andare al passo 5 altrimenti andare al passo 6.
5. Prendere a caso un arco non marcato, e marcarlo. Poi seguirlo fino al nuovo nodo corrente e andare al passo 3.
6. Si è giunti ad un punto morto: bisogna ritornare al nodo precedente, cioè quello che era attuale prima dell'ultimo esaminato, renderlo il nodo corrente e andare al passo 3. Se invece il nodo è quello di partenza, il grafo non contiene alcun ciclo e l'algoritmo termina.

Questo algoritmo considera, a turno, ogni nodo come radice di quello che ritiene sarà un albero, e fa una ricerca di tipo anticipato su di esso: se ritorna ad un nodo già incontrato allora ha trovato un ciclo; se visita tutti gli archi uscenti da un certo nodo, torna al nodo precedente. Se torna alla radice e non può più andare oltre, il sottografo raggiungibile dal nodo corrente non contiene cicli. Nel caso che questa proprietà sia verificata per tutti i nodi, l'intero grafo è senza cicli e il sistema non è in stallo.

Per vedere come opera questo algoritmo in pratica, usiamo il grafo di Figura 3.5(a): l'ordine con cui si visitano i nodi è arbitrario, per cui si decide di ispezionarli da sinistra verso destra, dall'alto al basso, facendo partire l'algoritmo da R e visitando successivamente A, B, C, S, D, T, E, F , e così via. L'algoritmo si ferma quando si incontra un ciclo.

Si parte da R e si inizializza L come lista vuota, si aggiunge R alla lista, ci si muove verso A , che è l'unico nodo raggiungibile, lo si aggiunge a L e si ha $L = [R, A]$. Da A si procede verso S , ottenendo $L = [R, A, S]$; S non ha archi uscenti, così forza l'algoritmo a tornare ad A ; poiché A non ha archi uscenti non marcato, si torna indietro a R , completando l'ispezione di R .

L'algoritmo riparte da A , reinizializzando L come lista vuota. Anche questa ricerca termina velocemente, per cui si riparte da B , continuando a seguire gli archi uscenti finché non si giunge a D e si ottiene $L = [B, T, E, V, G, U, D]$; si procede poi con una scelta (casuale): se si considera S si termina e si ritorna a D . La seconda volta si considera T e si aggiorna $L = [B, T, E, V, G, U, D, T]$, in questo momento si scopre un ciclo e l'algoritmo termina.

Questo algoritmo non è uno dei migliori, per studiarne uno migliore guardare (Even, 1979); ciononostante, la sua esistenza dimostra che ce n'è almeno uno per l'identificazione del deadlock.

3.4.2 Identificazione del deadlock con risorse multiple per ogni classe

Quando esistono più copie di alcune risorse è necessario un approccio diverso per identificare i deadlock. Vedremo un algoritmo basato su matrici per l'identificazione dei

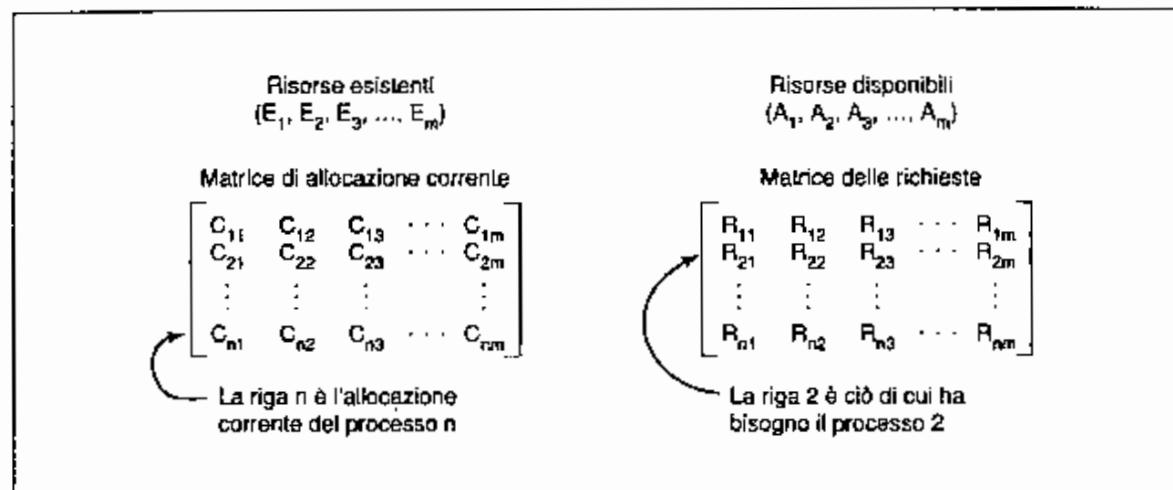


Figura 3.6 Le quattro strutture dati necessarie all'algoritmo di rilevazione del deadlock.

deadlock di n processi, da P_1 a P_n . Si supponga che il numero di classi di risorse sia m , con E_1 risorse di classe 1, E_2 risorse di classe 2, e generalmente E_i risorse di classe i ($1 \leq i \leq m$). E è il vettore di risorse esistenti che dà il totale di istanze per ogni risorsa esistente. Ad esempio, se la classe 1 è un dispositivo a nastro, allora $E_1 = 2$ significa che il sistema ha due dispositivi a nastro.

In ogni momento ci sono alcune risorse assegnate e non disponibili. Si supponga che A sia il vettore di risorse disponibili, con A_i che dà il numero di istanze della risorsa i che sono attualmente disponibili (ad esempio, non assegnate). Se i due dispositivi a nastro sono assegnati allora A_1 sarà 0.

Ora abbiamo bisogno di due array, C , la matrice di allocazione corrente, e R , la matrice di richiesta. L' i -esima riga di C dice quante istanze di ogni classe di risorse sono assegnate al processo P_i e C_{ij} indica il numero di risorse di classe j occupate dal processo i . Allo stesso modo, R_{ij} è il numero di risorse della classe j che P_i desidera; queste quattro strutture dati sono mostrate in Figura 3.6.

Per queste quattro strutture dati vale la regola invariante che ogni risorsa è disponibile o occupata. Questa osservazione implica che

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

In altre parole, se si sommano tutte le istanze della risorsa j che sono state allocate, ed a queste si aggiungono tutte le istanze che sono disponibili, il risultato è il numero di istanze esistenti di quella classe di risorse.

L'algoritmo di identificazione del deadlock si basa sul confronto fra vettori. Si definisca ora la relazione $A \leq B$ sui due vettori A e B per dire che ogni elemento di A è minore o uguale al corrispondente elemento di B . In termini matematici, $A \leq B$ è vero se e solo se $A_i \leq B_i$ per $1 \leq i \leq m$.

Inizialmente ogni processo si considera come non marcato; con l'avanzamento dell'algoritmo i processi saranno marcati, indicando che essi possono terminare e cioè che non sono in stallo. Quando l'algoritmo termina, qualsiasi processo non marcato risulta in stallo.

Si può ora descrivere l'algoritmo di identificazione del deadlock:

1. Cerca un processo non marcato, P_i , per cui l'-esima riga di R è minore uguale di A .
2. Se si trova un processo di questo tipo, aggiunge l'-esima riga di C ad A , marca il processo e torna al passo 1.
3. Se non esiste nessun processo di questo tipo l'algoritmo termina.

Al termine dell'algoritmo, se ci sono processi non marcati, questi sono in stallo.

Nel passo 1 l'algoritmo cerca un processo che possa essere eseguito e completato, cioè un processo caratterizzato dal richiedere risorse che si trovano tra quelle attualmente disponibili. Il processo selezionato è così eseguito finché termina e restituisce le risorse che stava occupando all'insieme delle risorse disponibili, dopodiché è marcato come terminato. Alla fine, se tutti i processi sono disponibili per l'esecuzione, allora nessuno di essi è in stallo. Se qualcuno di essi non è mai disponibile per l'esecuzione allora è in stallo. Sebbene l'algoritmo sia non-deterministico (poiché esso può eseguire i processi in un ordine qualsiasi), il risultato è sempre lo stesso.

Un esempio di come opera l'algoritmo di identificazione del deadlock è mostrato in Figura 3.7: si hanno tre processi e quattro classi di risorse, che arbitrariamente sono state etichettate come unità a nastro, plotter, scanner e unità CD ROM. Il processo 1 occupa uno scanner, il processo 2 occupa le due unità a nastro e l'unità CD ROM, il processo 3 occupa un plotter e due scanner. Ogni processo ha bisogno di una ulteriore risorsa come mostrato nella matrice R .

Per eseguire l'algoritmo di identificazione del deadlock si cerca un processo la cui richiesta di risorsa possa essere soddisfatta. La prima non può essere soddisfatta perché l'unità CD ROM non è disponibile, la stessa cosa accade per la seconda perché lo scanner non è libero. Fortunatamente la terza può essere soddisfatta, così il processo 3 va in esecuzione e prima o poi restituisce tutte le risorse. Si ha dunque

$$A = (2 \ 2 \ 2 \ 0)$$

A questo punto può partire il processo 2 e restituire le risorse. Si ha

$$A = (4 \ 2 \ 2 \ 1)$$

Ora i processi rimanenti possono iniziare l'esecuzione: nel sistema non c'è alcun deadlock.

Dispositivi a nastro				Dispositivi a nastro			
	Plotter	Scanner	CD Rom		Plotter	Scanner	CD Rom
$E = (4 \ 2 \ 3 \ 1)$				$A = (2 \ 1 \ 0 \ 0)$			
Matrice di allocazione corrente				Matrice delle richieste			
$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$				$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$			

Figura 3.7 Un esempio dell'algoritmo di identificazione del deadlock.

Si consideri ora una piccola variazione della situazione di Figura 3.7: supponiamo che il processo 2 abbia bisogno sia dell'unità CD ROM sia delle due unità a nastro e del plotter. Nessuna delle richieste può essere soddisfatta per cui l'intero sistema è in stallo.

Ora che si sa come rilevare i deadlock, sorge il problema di quale sia il momento giusto di cercarli. Una possibilità consiste nel controllare ogni volta che viene fatta una richiesta di risorsa: così sicuramente potremo scoprirli il più presto possibile, anche se ciò in termini di tempo di CPU è molto costoso. Una strategia alternativa consiste nel fare un controllo ogni k minuti, o magari solo quando l'utilizzo della CPU scende al di sotto di una determinata soglia. Il motivo per cui si considera l'utilizzo della CPU è che se un certo numero di processi sono in stallo, allora ci saranno pochi processi che potranno essere eseguiti, e la CPU sarà spesso inutilizzata.

3.4.3 La risoluzione del deadlock

Supponiamo che l'algoritmo di scoperta del deadlock abbia avuto successo e che abbia rilevato un deadlock. Quale sarà il prossimo passo? È necessaria qualche tecnica che permetta al sistema di ricominciare a funzionare. In questo paragrafo si discuteranno diverse tecniche di risoluzione del deadlock; tuttavia bisogna notare che nessuna di esse è particolarmente attraente.

La risoluzione del deadlock mediante prerilascio

In alcuni casi può essere possibile togliere una risorsa ad un processo e assegnarla ad un altro. In molti casi è richiesto un intervento manuale, specialmente nei sistemi operativi con elaborazione batch su grossi elaboratori.

Ad esempio, per togliere una stampante laser al processo in corso l'operatore può raccogliere tutti i fogli già stampati, sistemarli da una parte e sospendere il processo (marcandolo come non eseguibile). A questo punto la stampante può essere assegnata ad un altro processo. Quando questo termina, si può ristabilire la situazione precedente risistemando sulla stampante la pila di fogli già stampati dal primo processo facendolo ripartire.

Togliere una risorsa ad un processo assegnandola ad un altro e, al suo termine, riassegnarla al primo senza che questi se ne accorga dipende molto dal tipo di risorsa in oggetto: molto spesso, questa soluzione è impossibile o difficile. La scelta del processo da sospendere dipende molto da quali processi hanno risorse che si possono facilmente restituire.

Risoluzione del deadlock mediante ritorno a uno stato precedente

Se il progettista di sistema e gli operatori di macchina sapessero che i deadlock sono eventi probabili, essi potrebbero decidere di mettere periodicamente un punto di controllo (check-point) sui processi: ciò implica scrivere lo stato del processo in un file, in modo da poter ripartire in un momento successivo. Il punto di controllo contiene non solo l'immagine della memoria, ma anche lo stato delle risorse, cioè quali sono le risorse assegnate in quell'istante al processo. Per essere maggiormente efficace, un nuovo punto di controllo non dovrà essere riscritto su uno precedente, ma dovrà essere scritto su un nuovo file, in modo che l'avanzamento del processo presupponga l'accumularsi di un'intera sequenza di file di punti di controllo.

Quando si scopre un deadlock, è facile vedere quali sono le risorse coinvolte. Per risol-

vere il deadlock, un processo che possiede una risorsa richiesta è riportato a uno stato precedente al tempo in cui ha acquisito tale risorsa facendolo ripartire da uno dei suoi punti di controllo (rollback). Tutto il lavoro svolto dopo il punto di controllo è perduto (ad esempio, l'output stampato dopo il punto di controllo deve essere scartato, perché sarà stampato di nuovo). In effetti, si riporta il processo ad un momento precedente, in cui non aveva la risorsa, che ora è assegnata ad uno dei processi in stallo. Se il processo ripartito prova ad acquisire di nuovo la risorsa, dovrà attendere finché non sarà disponibile.

La risoluzione del deadlock mediante l'eliminazione del processo

Il modo brutale, ma più semplice, per risolvere un deadlock è quello di eliminare uno o più processi nel ciclo. Con un po' di fortuna, gli altri processi potranno continuare. Se ciò non fosse di aiuto, quanto detto precedentemente può essere ripetuto fino a quando non si interrompe il ciclo.

In alternativa si potrebbe scegliere attentamente come vittima un processo non appartenente al ciclo, per ottenere il rilascio delle sue risorse, che potrebbero servire a qualche processo nel ciclo. Ad esempio, un processo sta occupando una stampante e vuole un plotter, mentre un altro processo sta occupando un plotter ed è in attesa di una stampante: questi due processi sono in stallo. Un terzo processo può occupare un'altra stampante ed un altro plotter, ed essere in esecuzione senza problemi: eliminando il terzo processo si interromperà il deadlock che coinvolge i primi due.

Dove possibile, è meglio eliminare un processo che può essere rieseguito dall'inizio senza effetti dannosi. Ad esempio, una compilazione può essere sempre rieseguita perché si limita a leggere un file sorgente e a produrre un file oggetto. Se essa fosse interrotta a metà, la prima esecuzione non avrebbe alcuna influenza sulla seconda.

D'altra parte un processo che aggiorna un database non sempre può essere rieseguito una seconda volta senza correre pericoli. Se il processo somma 1 a qualche record nel database, eseguendolo una volta, eliminandolo e poi eseguendolo di nuovo, somma 2 al record, e ciò non è corretto.

3.5 Evitare i deadlock

Nel discutere l'identificazione del deadlock si è supposto tacitamente che quando un processo richiede delle risorse le richiede tutte insieme (la matrice R di Figura 3.6). In molti sistemi, invece, le risorse sono richieste una alla volta, ed il sistema deve poter decidere se assegnare una risorsa è un'azione sicura e fare l'assegnazione solo in questo caso. Da questa considerazione nasce una domanda: "Esiste un algoritmo che evita sempre il deadlock facendo ogni volta la scelta più opportuna?" La risposta è affermativa: si possono evitare i deadlock, ma solo se certe informazioni sono disponibili in anticipo. In questa sezione esamineremo alcuni modi per evitare il deadlock allocando le risorse con attenzione.

3.5.1 Traiettorie di risorsa

Gli algoritmi principali per evitare il deadlock si basano sul concetto degli stati sicuri. Prima di descrivere gli algoritmi si parlerà brevemente del problema della sicurezza utilizzando un disegno che ne renderà più facile la comprensione. Sebbene l'approccio grafico non si possa tradurre direttamente in un algoritmo, esso permette di capire abbastanza bene ed in modo intuitivo il problema.

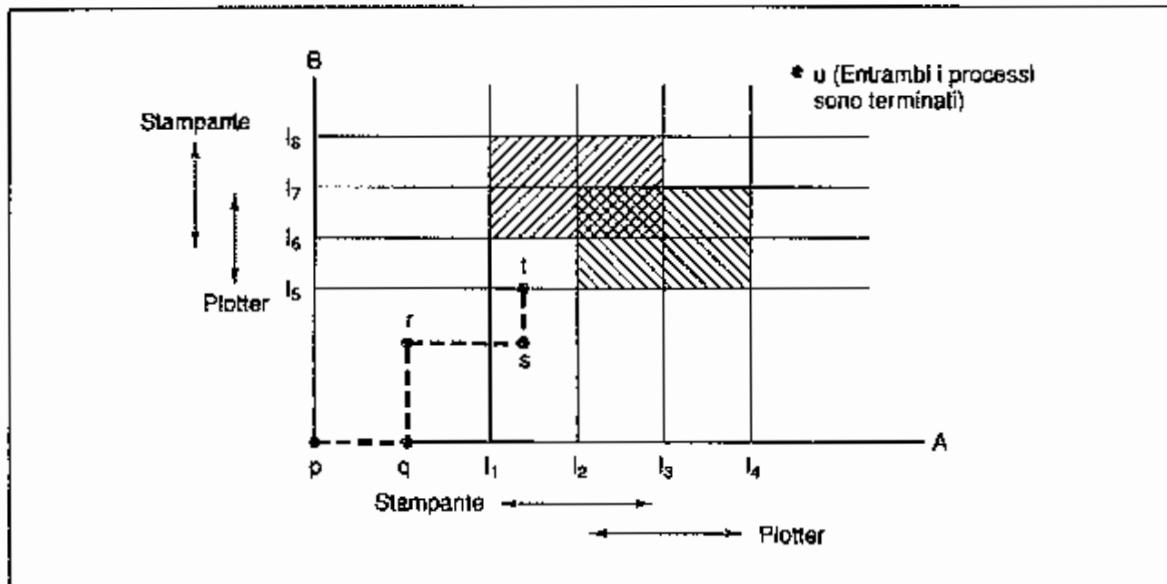


Figura 3.8 Traiettorie di risorsa per due processi.

In Figura 3.8 si vede un modello in grado di trattare due processi e due risorse: una stampante ed un plotter, ad esempio. L'asse orizzontale rappresenta il numero di istruzioni eseguite dal processo A; l'asse verticale rappresenta il numero di istruzioni eseguite dal processo B. In I_1 , A richiede una stampante; in I_2 , ha bisogno di un plotter. La stampante ed il plotter sono rilasciati in I_3 e in I_4 , rispettivamente; il processo B ha bisogno del plotter da I_5 a I_7 , e della stampante da I_6 a I_8 .

Ogni punto nel diagramma rappresenta uno stato congiunto dei due processi. Inizialmente, lo stato è in p , e nessuno dei due processi ha eseguito alcuna istruzione. Se lo scheduler sceglie A per farlo eseguire per primo, si arriva al punto q , nel quale A ha eseguito un certo numero di istruzioni e B non ne ha eseguita nessuna; nel punto q , la traiettoria diventa verticale, indicando che lo scheduler ha scelto di eseguire B. Se si ha un singolo processore, tutti i camminii devono essere orizzontali o verticali, mai diagonali; inoltre, lo spostamento avviene sempre verso nord o verso est e mai verso sud o verso ovest (i processi non possono ritornare indietro).

Quando A attraversa la linea I_1 sul cammino da r a s , significa che esso ha richiesto ed ottenuto la stampante; quando B raggiunge il punto t , esso richiede il plotter.

Le regioni di piano tratteggiate sono particolarmente interessanti: la regione tratteggiata con le linee da nord-ovest verso sud-est, rappresenta lo stato in cui entrambi i processi detengono la stampante, e la regola di mutua esclusione fa sì che sia impossibile entrare in questa zona. In maniera analoga, la regione tratteggiata in maniera opposta rappresenta lo stato in cui tutti e due i processi hanno il plotter, il che è nuovamente impossibile.

Se il sistema entrasse nella scatola delimitata da I_1 e I_2 sui lati verticali e da I_5 e I_6 sopra e sotto, esso si bloccherebbe sull'incrocio di I_6 con I_2 . A questo punto, A sta richiedendo il plotter e B la stampante, ma entrambe queste risorse sono già state assegnate: l'intera scatola non è sicura e quindi non vi si deve mai entrare. Al punto t , la sola cosa sicura che si può fare è di far eseguire il processo A finché arriva a I_4 . Dopo questo punto, una qualunque traiettoria che arrivi ad u può essere scelta con successo.

La cosa importante da notare è che, nel punto t , B sta richiedendo una risorsa. Il sistema deve decidere se assegnarla o meno, ma se la assegna, il sistema entrerà in una regio-

ne non sicura e prima o poi andrà in deadlock. Per evitare il deadlock *B* deve essere sospeso finché *A* non abbia richiesto e rilasciato il plotter.

3.5.2 Stati sicuri e non sicuri

Gli algoritmi per evitare il deadlock che si studieranno usano l'informazione di Figura 3.6: in ogni istante c'è uno stato corrente che è formato da *E*, *A*, *C* e *R*. Uno stato è detto **sicuro** se non è in stallo, e se c'è un modo per soddisfare tutte le richieste pendenti eseguendo i processi in un qualche ordine. Questo concetto si illustra più facilmente tramite un esempio che usa una risorsa: in Figura 3.9(a) si ha uno stato in cui *A* ha 3 istanze della risorsa e può richiederne fino a 9; *B* ne ha 2 ma può richiederne fino a 4 in un secondo momento; e *C* ne ha 2 ma può richiederne 5 in più. Esiste un totale di 10 istanze della risorsa, di cui 7 sono allocate e 3 sono libere.

Lo stato di Figura 3.9(a) è sicuro perché esiste una sequenza di allocazioni che permette a tutti i processi di terminare. In effetti, lo scheduler potrebbe eseguire esclusivamente *B*, finché esso richiede e prende altre due istanze della risorsa passando allo stato della Figura 3.9(b). Quando *B* termina, si giunge nello stato di Figura 3.9(c), dopodiché lo scheduler può eseguire *C*, passando dopo qualche tempo nello stato di Figura 3.9(d). Quando termina *C* si ha lo stato di Figura 3.9(e); ora *A* può avere le sei istanze della risorsa di cui ha bisogno e terminare, per cui lo stato di Figura 3.9(a) è sicuro perché il sistema, tramite un'opportuna schedulazione, può evitare il deadlock.

Ora supponiamo di avere lo stesso stato iniziale come mostrato in Figura 3.10(a), ma questa volta *A* richiede e prende un'altra risorsa, come mostrato in Figura 3.10(b). Si può trovare una sequenza che permetta di lavorare? Proviamoci. Lo scheduler potrebbe eseguire *B* fino al massimo delle risorse che ha richiesto (Figura 3.10(c)) e poi rilasciarle ed arrivare alla situazione di Figura 3.10(d).

A questo punto si è bloccati. Si hanno solo quattro istanze libere della risorsa ed ognuno dei processi attivi ne richiede 5: non c'è nessuna sequenza che garantisce la terminazione. Tutto ciò perché la decisione di allocazione che ha portato il sistema da Figura 3.10(a) a Figura 3.10(b) l'ha portato da uno stato sicuro a uno stato non sicuro; nemmeno eseguire *A* o *C*, dopo essere partiti da Figura 3.10(b), funziona. A posteriori, la richiesta di *A* non si sarebbe dovuta soddisfare.

Val la pena notare che uno stato non sicuro non è in stallo: partendo da Figura 3.10(b), il sistema può eseguire per un certo tempo, ed almeno un processo può essere completato. Inoltre è possibile che *A* rilasci una risorsa prima di chiederne un'altra, permettendo

	Possedute	Massimo												
A	3	9		A	3	9		A	3	9		A	3	9
B	2	4		B	4	4		B	0	-		B	0	-
C	2	7		C	2	7		C	2	7		C	7	7
Libere: 3 (a)			Libere: 1 (b)			Libere: 5 (c)			Libere: 0 (d)			Libere: 7 (e)		

Figura 3.9 Dimostrazione che lo stato di (a) è sicuro.

	Possedute	Massimo									
A	3	9	A	4	9	A	4	9	A	4	9
B	2	4	B	2	4	B	4	4	B	-	-
C	2	7	C	2	7	C	2	7	C	2	7
Libere: 3			Libere: 2			Libere: 0			Libere: 4		
(a)			(b)			(c)			(d)		

Figura 3.10 Dimostrazione che lo stato di (b) non è sicuro.

a C di terminare ed evitando lo stallo. Per cui la differenza tra uno stato sicuro e uno stato non sicuro è che in uno stato sicuro il sistema può garantire che tutti i processi avranno termine, mentre in uno stato non sicuro non c'è nessuna garanzia di questo tipo.

3.5.3 L'algoritmo del banchiere per una singola risorsa

L'algoritmo del banchiere dovuto a Dijkstra (1965) è un algoritmo di scheduling che permette di evitare le situazioni di stallo ed è un'estensione dell'algoritmo di identificazione del deadlock, dato nella Sezione 3.4.1. Esso prende spunto dalla maniera in cui un banchiere di una piccola città tratta un gruppo di clienti ai quali ha garantito delle linee di credito. Quello che l'algoritmo fa è controllare se il garantire una richiesta porti ad uno stato non sicuro e se ciò accade, la richiesta viene rifiutata. In Figura 3.11(a) si vedono quattro clienti A, B, C e D, ognuno dei quali ha avuto un certo numero di unità di credito (per esempio un'unità di credito è pari a 1000 dollari). Il banchiere sa che non tutti i clienti avranno bisogno immediatamente del loro credito massimo, così ha riservato, per servirli, solo 10 unità invece di 22 (in questo esempio, i clienti rappresentano i processi, le unità di credito delle unità a nastro e il banchiere è il sistema operativo).

I clienti gestiscono i loro affari richiedendo di tanto in tanto dei prestiti (cioè chieden-

	Possedute	Massimo							
A	0	6	A	1	6	A	1	6	
B	0	5	B	1	5	B	2	5	
C	0	4	C	2	4	C	2	4	
D	0	7	D	4	7	D	4	7	
Libere: 10			Libere: 2			Libere: 1			
(a)			(b)			(c)			

Figura 3.11 Tre stati di allocazione delle risorse: (a) Sicuro, (b) Sicuro, (c) Non sicuro.

do delle risorse). La situazione, in un certo istante, è descritta dalla Figura 3.11(b): questo stato è sicuro poiché il banchiere, con due unità ancora da assegnare, può ritardare tutte le richieste tranne quella di C; così facendo C può finire e restituire tutte le sue 4 risorse. Con altre quattro unità disponibili il banchiere può far sì che D o B abbiano le unità necessarie e così via.

Si consideri cosa sarebbe successo se un'eventuale richiesta di ulteriori unità da parte di B fosse stata accolta nello stato della Figura 3.11(b). Si sarebbe avuta la situazione della Figura 3.11(c), che non rappresenta uno stato sicuro. Se tutti i clienti richiedessero la cifra massima per i loro prestiti immediatamente, il banchiere non sarebbe in grado di soddisfare nessuno di loro, e in questo caso si avrebbe una situazione di stallo. Uno stato non sicuro *non deve* necessariamente portare ad un deadlock, poiché il cliente può non richiedere immediatamente tutta la sua linea di credito, ma il banchiere non può contare sul fatto che si verifichi proprio un simile comportamento.

L'algoritmo del banchiere considera così ogni richiesta nel momento in cui essa si presenta, ed analizza se soddisfarla porta ad uno stato sicuro. Se lo fa, soddisfa la richiesta, altrimenti la richiesta viene posposta. Per vedere se uno stato è sicuro, il banchiere controlla di avere abbastanza risorse a disposizione per poter soddisfare le richieste di un cliente. Se è così, questo prestito può essere considerato restituito, e si può controllare il cliente che si trova ora più vicino al massimo della propria linea di credito, eccetera. Se tutti i prestiti possono essere restituiti, lo stato è sicuro e la richiesta iniziale può essere soddisfatta.

3.5.4 L'algoritmo del banchiere per risorse multiple

L'algoritmo del banchiere può essere generalizzato per trattare anche risorse multiple. La Figura 3.12 mostra come funziona l'algoritmo in questo caso.

Nella Figura 3.12 si vedono due matrici: quella sulla sinistra mostra quante risorse sono state attualmente assegnate ad ognuno dei cinque processi, mentre la matrice a destra indica quante risorse sono ancora necessarie ad ogni processo per completare l'esecuzione. Queste matrici sono proprio le matrici C ed R di Figura 3.6. Come nel caso di una singola risorsa, i processi devono comunicare il loro fabbisogno globale di risorse prima di iniziare l'esecuzione, in modo tale che il sistema possa calcolare la matrice di destra in ogni istante.

	Processo	Dispositivi a nastro	Plotter	Scanner	CD ROM
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Risorse assegnate

	Processo	Dispositivi a nastro	Plotter	Scanner	CD ROM
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Risorse di cui si ha ancora bisogno

Figura 3.12 L'algoritmo del banchiere con risorse multiple.

I tre vettori sulla destra della figura indicano le risorse esistenti E , le risorse assegnate P , e le risorse disponibili A . Da E si può dedurre che il sistema ha sei unità a nastro, tre plotter, quattro stampanti e due unità CD ROM. Di questi, cinque unità a nastro, tre plotter, due stampanti e due unità CD ROM sono attualmente già stati assegnati. Questo si ricava sommando le quattro colonne delle risorse alla matrice di sinistra. Il vettore delle risorse disponibili è semplicemente la differenza fra le risorse disponibili nel sistema e quelle attualmente in uso.

Ora si può vedere come funziona l'algoritmo per controllare se un certo stato è sicuro.

1. Si cerchi una riga R per la quale le risorse ancora non richieste sono meno o altrettante di quelle in A . Se non esiste una riga di questo tipo, il sistema andrà in stallo perché nessun processo può essere eseguito e terminare.
2. Si supponga che il processo della riga scelta richieda tutte le risorse di cui ancora ha bisogno (e ciò è possibile per il punto 1) e finisca il proprio lavoro. Si marchi allora il processo come terminato e si aggiungano al vettore A tutte le risorse che deteneva.
3. Si ripetano i punti 1 e 2 fino a quando o tutti i processi vengono marcati, e quindi si può affermare che lo stato di partenza era sicuro, oppure ci si trova in una situazione di stallo, nel qual caso lo stato di partenza non era sicuro.

Nel caso in cui si possa scegliere più di un processo durante il passo 1, l'ordine con cui la scelta viene effettuata non ha importanza; infatti l'insieme delle risorse disponibili non può che aumentare o, nel caso peggiore, rimanere lo stesso.

Ora ritorniamo all'esempio della Figura 3.12: lo stato corrente è sicuro; si supponga che il processo B richieda una stampante. Questa richiesta può essere soddisfatta perché lo stato che si raggiunge è a sua volta uno stato sicuro (il processo D può terminare, poi possono terminare i processi A ed E , ed infine possono terminare tutti i restanti processi).

Si immagini poi che, dopo aver assegnato una delle due stampanti rimanenti a B , E richieda l'ultima stampante. Se la richiesta fosse soddisfatta, il vettore delle risorse disponibili diventerebbe $(1\ 0\ 0\ 0)$ e si avrebbe una situazione di stallo. Chiaramente, la richiesta del processo E deve essere ritardata per un po' per evitarlo.

Questo algoritmo è stato pubblicato da Dijkstra nel 1965, e da allora è stato descritto dettagliatamente in quasi tutti i libri di sistemi operativi, e sono stati scritti molti articoli sui suoi vari aspetti. Sfortunatamente, pochi autori hanno avuto il coraggio di far notare come, sebbene sia un algoritmo ottimo in teoria, esso è privo di ogni utilità pratica per via del fatto che raramente i processi sono in grado di stabilire in anticipo il massimo numero di risorse di cui hanno bisogno per completare l'esecuzione. Oltre a ciò, il numero dei processi non è di solito fisso, ma varia dinamicamente a seconda degli utenti che si connettono al sistema. Infine, risorse che vengono ritenute disponibili ad un certo istante possono non esserlo più all'istante successivo (ad esempio, una unità a nastro potrebbe rompersi). In pratica pochi, o forse nessuno, dei sistemi operativi esistenti usano l'algoritmo del banchiere per evitare i deadlock.

3.6 Prevenzione da deadlock

Come si è visto, evitare il deadlock è praticamente impossibile, perché bisogna avere informazioni sulle richieste future, che non si conoscono. Come fa il sistema, di fatto, ad evitare il deadlock? Andiamo a vedere se le quattro condizioni indicate da Coffmann ed altri (1971) forniscono una traccia per possibili soluzioni. Se si potesse essere sicuri che

almeno una delle condizioni non sarà mai verificata, allora non si avrebbero mai situazioni di stallo (Havender 1968).

3.6.1 Negazione della condizione di mutua esclusione

Prendiamo in considerazione la condizione di mutua esclusione: se nessuna risorsa fosse mai assegnata in maniera esclusiva ad un unico processo, non avremmo mai situazioni di stallo. D'altra parte, è altrettanto chiaro che permettere a due processi di scrivere contemporaneamente su di una stampante porterebbe al caos. Con lo spooling dell'uscita su stampante, più processi potrebbero generare stampe nello stesso istante. In questo modello, l'unico processo che richiede fisicamente la stampante è il demone di stampa e, poiché il demone di stampa non richiederà mai altre risorse, per la stampante il problema del deadlock è eliminato.

Sfortunatamente, non tutti i dispositivi possono venire trattati con la tecnica dello spooling (ad esempio, la tabella dei processi non potrà mai essere gestita in spooling). Inoltre, la competizione che si genera per ottenere lo spazio su disco necessario per lo spooling può di per sé portare ad una situazione di deadlock. Che cosa succederebbe se due processi riempissero ciascuno metà dello spazio di spool disponibile senza che nessuno dei due abbia finito il proprio lavoro? Se il demone della stampante fosse stato programmato per iniziare a stampare prima che tutto il file di spool sia già scritto sul disco, la stampante potrebbe rimanere inattiva, se un processo di uscita decidesse di aspettare ore dopo una prima sequenza di stampe. Per questa ragione i demoni sono normalmente programmati per stampare solo dopo che è disponibile l'intero file che deve essere stampato. In questo caso abbiamo due processi che hanno terminato in parte, ma non del tutto, il loro output e non possono continuare. Nessuno dei processi finirà mai, e così avremmo una situazione di stallo relativa al disco.

Nondimeno, ora c'è un'idea che si può applicare spesso: evitare di assegnare una risorsa quando non è strettamente necessario, ed assicurarsi che pochi processi, il minor numero possibile, esigano la risorsa.

3.6.2 Negazione della condizione hold-and-wait

La seconda delle condizioni enunciate da Coffman sembra più promettente. Se si potesse evitare che un processo che già detiene delle risorse ne richieda della altre, si potrebbero eliminare i deadlock. Un modo per raggiungere questo obiettivo è di obbligare tutti i processi a richiedere tutte le risorse di cui hanno bisogno prima di iniziare l'esecuzione. Se tutte le risorse richieste risultano disponibili, allora il processo potrà essere mandato in esecuzione, avrà tutto quello di cui ha bisogno, e potrà essere eseguito fino alla propria fine logica. Al contrario, se una o più risorse sono occupate, il processo non viene mandato in esecuzione, non gli si alloca nulla, e rimane in attesa.

Un primo problema che si pone in questo approccio è che molti processi non sanno di quante risorse hanno bisogno finché non cominciano l'esecuzione: se lo sapessero, si potrebbe usare l'algoritmo del banchiere. Un secondo problema è rappresentato dal fatto che in questa maniera non viene ottimizzato l'uso delle risorse. Si prenda, ad esempio, un processo che legge dati da un nastro di ingresso, li analizza per un'ora e poi scrive i risultati su un nastro di uscita e su un plotter. Se tutte le risorse devono essere richieste anticipatamente, il processo terrà occupato il nastro di uscita ed il plotter per un'ora, inutilmente.

Comunque, alcuni sistemi batch di grossi calcolatori chiedono all'utente di elencare tutte le risorse di cui ha bisogno sulla prima riga di ogni job; in questo modo il sistema acquisisce tutte le risorse immediatamente e le tiene occupate finché non è terminato il

job. Questo metodo appesantisce il lavoro del programmatore e spreca le risorse, però previene i deadlock.

Una maniera leggermente differente di interrompere la condizione di hold-and-wait (prendi e aspetta), è quella di fare in modo che un processo che richiede una risorsa rilasci temporaneamente tutte le risorse che deteneva in quel momento, e poi richieda tutte le risorse contemporaneamente.

3.6.3 Negazione della condizione di mancanza di prerilascio

Prendere in considerazione la terza condizione (mancanza di prerilascio) risulta essere ancora meno promettente del prendere in considerazione la seconda. Se ad un processo è stata assegnata la stampante, ed esso si trova nel mezzo di una stampa, togliergli la stampante di forza perché un plotter non è disponibile richiede dei trucchi (quando va bene) oppure è impossibile (quando va male).

3.6.4 Negazione della condizione di attesa circolare

Rimane una sola condizione: l'attesa circolare può essere eliminata in diverse maniere. Si può semplicemente impostare una regola che faccia in modo che ogni processo abbia assegnata ad ogni istante una sola risorsa. Nel caso in cui un processo abbia bisogno di una seconda risorsa, esso deve rilasciare la prima, prima di ottenerla. Nel caso di un processo che debba copiare un file molto lungo da un nastro ad una stampante, questa restrizione è chiaramente inaccettabile.

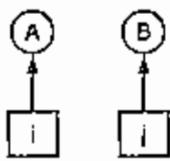
Un altro metodo per evitare l'attesa circolare è di fornire una numerazione globale a tutte le risorse presenti, come in Figura 3.13(a). Poi si adotta la seguente regola: i processi possono richiedere le risorse quando vogliono, ma tutte le richieste devono essere fatte seguendo l'ordinamento numerico. Un processo può richiedere prima una stampante e poi un lettore di nastro, ma non può richiedere un plotter prima di una stampante.

Con questa regola, il grafo di allocazione non può contenere cicli. Vediamo perché questo vale, nel caso si abbiano due processi come in Figura 3.13(b). Potremmo avere un deadlock solo se il processo A richiedesse la risorsa j e B richiedesse i . Supponendo che i e j siano risorse distinte, esse avranno un numero differente. Se $i > j$ allora A non può richiedere j ; se $i < j$ allora B non può richiedere i . In ogni caso è impossibile avere una situazione di stallo.

Nel caso si abbiano più processi vale la stessa logica: in ogni istante, una delle risorse assegnate possiede il numero più alto; il processo che detiene questa risorsa non potrà

1. Stampante a colori
2. Scanner
3. Plotter
4. Dispositivo a nastro
5. Unità CD Rom

(a)



(b)

Figura 3.13 (a) Risorse ordinate numericamente. (b) Un grafo delle risorse.

Condizione	Approccio
Mutua esclusione	Spool di tutto
Hold and wait	Richiedere inizialmente tutte le risorse
No prelascio	Portare via le risorse
Attesa circolare	Ordinare numericamente le risorse

Figura 3.14 Sommario degli approcci alla prevenzione dei deadlock.

mai richiedere una risorsa già assegnata e quindi esso finirà oppure richiederà delle altre risorse, di numero ancora più alto, che sono comunque disponibili; ad un certo punto terminerà e rilascerà tutte le risorse che deteneva. A questo punto esisterà un altro processo che detiene la risorsa occupata con il numero più alto, ed esso potrà terminare per le stesse considerazioni. In breve, esiste uno scenario in cui tutti i processi terminano e non si riscontrano situazioni di stallo.

Una piccola variante di questo algoritmo è quella di rilasciare il vincolo della richiesta delle risorse in ordine strettamente crescente, ma di continuare a richiedere che nessun processo possa richiedere una risorsa minore (come numerazione) di quelle che sta già detenendo. Se un processo inizialmente richiede le risorse 9 e 10 e poi le rilascia entrambe, è come se effettivamente stesse cominciando da capo, e non c'è ragione di proibirgli di richiedere la risorsa 1.

Sebbene ordinare le risorse numericamente elimini il problema dei deadlock, può risultare impossibile trovare un ordinamento che sia soddisfacente per tutti. Quando le risorse includono tabelle dei processi, spazio di spool sul disco, basi di dati con record bloccati e altre risorse di tipo astratto, il numero delle risorse potenziali e dei differenti utenti può essere tanto grande che nessun ordinamento potrebbe funzionare.

I vari approcci alla prevenzione dei deadlock sono riassunti nella Figura 3.14.

3.7 Altri argomenti

In questo sezione si parlerà di altri argomenti relativi ai deadlock, tra cui il blocco a due fasi (two-phase locking), il deadlock senza risorse (non-resource deadlock) e la mancanza di risorse (starvation).

3.7.1 Il blocco a due fasi

Sebbene evitare o prevenire il deadlock non sia molto efficace in generale, per applicazioni specifiche si conoscono molti algoritmi eccellenti. Ad esempio, in molti sistemi di basi di dati, un'operazione che si presenta frequentemente è la richiesta di lock su diversi record e successivamente l'aggiornamento di tutti i record bloccati. Quando più processi stanno girando contemporaneamente c'è un pericolo reale di avere stallo.

L'approccio usato spesso è chiamato **blocco a due fasi** (two-phase locking). Nella prima fase il processo prova a bloccare tutti i record di cui ha bisogno, uno alla volta. Se ci riesce allora inizia la seconda fase eseguendo gli aggiornamenti e rilasciando i lock; nella prima fase non fa nessun lavoro reale.

Se durante la prima fase si ha bisogno di qualche record che è stato bloccato, il processo rilascia semplicemente tutti i suoi lock ed inizia nuovamente la prima fase. In un

certo senso questo approccio è come richiedere tutte le risorse in anticipo, o almeno prima che sia fatto qualcosa di irreversibile. In qualche versione del blocco a due fasi non c'è il rilascio e il riavvio se si incontra un lock durante la prima fase; in queste versioni si può avere il deadlock.

Comunque questa strategia non è applicabile al caso generale. Nei sistemi in tempo reale e in quelli di controllo di processo, ad esempio, non è accettabile terminare un processo in corso di esecuzione perché una risorsa non è disponibile, e ricominciare tutto dal principio. Non è accettabile ripartire di nuovo neanche quando il processo ha letto o scritto un messaggio su una rete, ha aggiornato file o ha eseguito qualunque altra operazione che non si può ripetere in modo sicuro. L'algoritmo funziona solo in quelle situazioni in cui il programmatore ha sistemato le cose in modo appropriato facendo sì che il programma possa essere interrotto in qualsiasi punto durante la prima fase e fatto ripartire. Diverse applicazioni, comunque, non si possono strutturare in questo modo.

3.7.2 Stallo senza risorse

Tutto ciò che si è visto riguardava i deadlock causati dalle risorse: un processo richiede qualcosa che sta usando un altro processo e deve attendere finché ciò è disponibile. I deadlock si possono comunque avere anche in altre situazioni che non coinvolgono completamente le risorse.

Ad esempio può accadere che due processi vadano in stallo aspettando che uno dei due faccia qualcosa, ciò accade spesso con i semafori. Nel Capitolo 2 si è parlato di esempi in cui un processo deve fare una down su due semafori, tipicamente *mutex* e su un altro semaforo. Se ciò è fatto nell'ordine sbagliato si può avere un deadlock.

3.7.3 Starvation

Un problema strettamente in relazione con il deadlock è la starvation: in un sistema dinamico le richieste per le risorse si fanno sempre, per cui è necessaria qualche politica che stabilisca l'ordine ed il tempo di utilizzo delle risorse. Sebbene apparentemente ragionevole, questa politica può far sì che alcuni processi non siano mai serviti pur non essendo in stallo.

Ad esempio, si consideri l'allocazione della stampante. Si immagini che il sistema usi qualche tipo di algoritmo per assicurare che l'allocazione della stampante non conduca a situazioni di stallo. Ora si supponga che più processi ne abbiano bisogno: a quali di essi sarà assegnata?

Un possibile algoritmo di allocazione consiste nell'assegnarla al processo con il file più piccolo da stampare (supponendo che questa informazione sia disponibile): questo approccio massimizza il numero di utenti soddisfatti e sembra equo. Ora si consideri cosa succede quando un processo ha un file molto grande da stampare: ogni volta che la stampante è libera, il sistema analizzerà e sceglierà il processo con il file più corto. Se c'è un flusso costante di processi con file corti, il processo con il file grande non usufruirà mai della stampante. In altre parole, "morirà d'inedia" (starvation) in attesa della risorsa, in quanto sarà posposto indefinitivamente anche se non è bloccato.

La mancanza di risorse si può evitare usando la politica di allocazione della risorsa Primo-arrivato, primo-servito (FIFO). Con questo approccio il processo che attende da più tempo sarà il prossimo ad essere servito: in un ragionevole lasso di tempo, ogni processo diventerà necessariamente il più vecchio, ed otterrà quindi la risorsa necessaria.

3.8 Ricerca sui deadlock

Se esiste un argomento che è stato studiato spietatamente durante i primi giorni dei sistemi operativi, questo è il deadlock: la ragione di ciò risiede nel fatto che l'identificazione del deadlock è un bel problemino di teoria di grafi, che un dottorando incline alla matematica può masticare per tre o quattro anni. Sono stati escogitati tutti i generi di algoritmo, ognuno più esotico e meno pratico del precedente; in poche parole, tutta questa ricerca si è estinta, con un articolo che appare ogni tanto (ad esempio, Karacali et al., 2000). Quando un sistema operativo vuole rilevare o impedire un deadlock, cosa che pochi fanno, utilizza uno dei metodi descritti in questo capitolo.

Tuttavia esiste ancora una minima ricerca sulla rilevazione del deadlock distribuito, ma non lo discuteremo in questo libro perché (1) rimane al di fuori degli scopi di questo libro e (2) non esiste ancora nulla di anche remotamente pratico nei sistemi reali. La sua funzione principale sembra quella di tenere lontani dalle strade i teorici dei grafi, altrimenti disoccupati.

3.9 Sommario

Il deadlock è un problema che potenzialmente si può presentare in ogni sistema operativo: si ha quando c'è un gruppo di processi in cui ognuno ha l'accesso esclusivo a qualcuna delle risorse, ed ognuno necessita di un'altra risorsa che appartiene ad un altro processo del gruppo. Tutti i processi sono bloccati e nessuno sarà mai in grado di ripartire.

Il deadlock si può evitare tenendo traccia degli stati sicuri e di quelli non sicuri: uno stato è sicuro quando esiste una sequenza di eventi che garantisce che tutti i processi possano terminare, non è sicuro quando non esiste questa garanzia. L'algoritmo del banchiere evita il deadlock non servendo una richiesta, se essa porterà il sistema in uno stato non sicuro.

Il deadlock si può prevenire strutturalmente progettando il sistema in modo che non si verifichi mai: ad esempio, permettendo ad un processo di occupare in ogni istante solo una risorsa alla volta, la condizione di attesa circolare richiesta per il deadlock non viene a verificarsi. Il deadlock si può prevenire anche numerando tutte le risorse, facendo sì che i processi le richiedano in ordine strettamente crescente. La starvation si può evitare con una politica di allocazione del tipo First-In First-Out.

PROBLEMI

1. Dare un esempio di stallo preso dalla politica.
2. Gli studenti che lavorano su PC individuali in un laboratorio informatico spediscono i loro file da stampare tramite un server, che effettua lo spool dei file sul suo hard disk. In quali condizioni può comparire un deadlock, nel caso in cui lo spazio su disco riservato allo spool sia limitato? Come si può evitare il deadlock?
3. Quali risorse della precedente domanda sono prerilasciabili e quali sono non prerilasciabili?
4. In Figura 3.1 le risorse vengono restituite in ordine inverso rispetto alla loro acquisizione. Sarebbe corretto restituirlle in un altro ordine?

5. La Figura 3.3 mostra il concetto di grafo di risorsa. Esistono grafi illegali, cioè grafi che violano strutturalmente il modello usato per l'utilizzo della risorsa? In caso affermativo, dare un esempio.
6. La discussione sull'algoritmo dello struzzo cita la possibilità di riempire le posizioni nella tabella dei processi o altre tabelle di sistema. Si può suggerire un modo in cui un amministratore di sistema possa uscire da una situazione del genere?
7. Si consideri la Figura 3.4. Si supponga che al passo (o) il processo C richieda S invece di R. Questa scelta può portare ad un deadlock? E se si suppone che richieda sia S sia R?
8. Ad un incrocio con uno STOP nelle quattro direzioni, la regola è dare la precedenza a chi viene da destra. Questa regola non è adeguata quando si avvicinano quattro veicoli in contemporanea; fortunatamente gli umani sono talvolta in grado di agire in maniera più intelligente dei computer ed il problema si risolve quando un autista segnala a quello alla sua sinistra di procedere. Si può descrivere una analogia tra questo comportamento ed uno dei metodi di prevenzione da stallo, descritti nella Sezione 3.4.3? Perché un problema di così facile soluzione nel mondo umano risulta così difficile da applicare ad un sistema computerizzato?
9. Si supponga che in Figura 3.6 $C_{ij} + R_{ij} > E_i$ per qualche valore di i . In questa ipotesi quali implicazioni si hanno per quei processi che terminano senza stallo?
10. Tutte le traiettorie della Figura 3.8 sono o verticali o orizzontali. Riuscite ad immaginare una situazione in cui è possibile avere traiettorie diagonali?
11. Si può usare l'algoritmo di traiettoria della risorsa di Figura 3.8 per illustrare il problema dei deadlock con tre processi e tre risorse? Nel caso fosse possibile, come si fa a realizzarlo? Nel caso non fosse possibile spiegare il motivo.
12. In teoria, l'algoritmo del grafo di traiettoria della risorsa si potrebbe usare per evitare i deadlock. Con una schedulazione intelligente, il sistema operativo potrebbe evitare le regioni non sicure. Esporre una situazione concreta in cui ciò avviene.
13. Osservate attentamente la Figura 3.11(b). Se D chiede un'altra unità, si raggiunge uno stato sicuro o uno non sicuro? Cosa succederebbe se la richiesta venisse da C invece che da D?
14. Un sistema può essere in uno stato che non è né di stallo né sicuro? Se sì, darne un esempio. Se no, dimostrare che uno stato è sempre uno stato di stallo oppure uno stato sicuro.
15. Un sistema ha due processi e tre risorse identiche. Ogni processo ha bisogno al massimo di due risorse. Può comparire il deadlock? Spiegare la propria risposta.
16. Considerare di nuovo il problema precedente, ma con p processi ognuno dei quali necessita di m risorse al massimo, ed un totale di r risorse disponibili. Quale condizione bisogna rispettare per far sì che il sistema sia privo di stallo?
17. Supponete che il processo A della Figura 3.12 richieda l'ultima unità a nastro. Così facendo si arriva ad una situazione di stallo?
18. Un calcolatore ha sei unità a nastro e ci sono n processi in competizione per usarli. Ogni processo ha bisogno di due nastri. Per quali valori di n il sistema non va mai in stallo?
19. L'algoritmo del banchiere viene eseguito su un sistema con m classi di risorse ed n

processi. Con m ed n molto grandi, il numero delle operazioni da eseguire per controllare se uno stato è sicuro è proporzionale a $m^a n^b$. Quali sono i valori di a e b ?

20. Un sistema ha quattro processi e cinque risorse allocabili. L'allocazione corrente e il massimo fabbisogno sono i seguenti:

	<i>Allocati</i>	<i>Massimo</i>	<i>Disponibile</i>
Processo A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Processo B	2 0 1 1 0	2 2 2 1 0	
Processo C	1 1 0 1 0	2 1 3 1 0	
Processo D	1 1 1 1 0	1 1 2 2 1	

Qual è il più piccolo valore di x per il quale questo è uno stato sicuro?

21. Un sistema distribuito che usa delle mailbox, ha due primitive IPC (Inter Process Communication): **send** e **receive**. Quest'ultima specifica il processo dal quale si vuole ricevere un messaggio e si blocca se non è disponibile alcun messaggio da quel processo, anche nel caso in cui siano disponibili messaggi da altri mittenti. Non ci sono risorse condivise, ma i processi hanno bisogno di comunicare frequentemente per altri scopi. È possibile avere una situazione di stallo? Discutere.
22. Due processi, *A* e *B*, necessitano ciascuno di tre record in un database, 1, 2, e 3. Se *A* li richiede con ordine 1, 2, 3 e *B* nello stesso ordine, non è possibile il deadlock. Tuttavia se *B* li richiede con ordine 3, 2, 1, allora il deadlock è possibile. Con 3 risorse, vi sono $3! = 6$ possibili combinazioni con cui ogni processo può richiedere le risorse. Quale frazione di tutte le combinazioni garantisce che non si avrà stallo?
23. Considerare il problema di prima, utilizzando però il blocco a due fasi. Verrà eliminato il pericolo potenziale di stallo? Ha tuttavia qualche caratteristica indesiderabile? Se sì, quali?
24. In un sistema di trasferimento elettronico di fondi, vi sono centinaia di processi identici che lavorano nella seguente maniera: ogni processo legge una linea di input che specifica l'ammontare del trasferimento, il conto sul quale deve essere accreditato, e quello dal quale deve essere prelevato l'ammontare. Poi blocca i due conti, e trasferisce la somma, rilasciando i lock immediatamente dopo. Quando si hanno tanti processi che girano in parallelo, esiste il pericolo che, avendo bloccato il conto *x*, non si riesca a bloccare il conto *y* perché è già bloccato da un altro processo che sta cercando di bloccare anche *x*. Ideare uno schema per evitare i deadlock. Non rilasciare il record di un conto finché non si completa la transazione. (In altre parole, non sono considerate valide le soluzioni che bloccano un conto e poi lo rilasciano immediatamente se l'altro è già bloccato.)
25. Un modo per impedire il deadlock è eliminare la condizione hold-and-wait. Nel testo si proponeva che un processo, prima di richiedere una nuova risorsa, dovesse prima rilasciare qualsiasi risorsa avesse già acquisito (supponendo che fosse possibile). Tuttavia facendo ciò si corre il pericolo di poter ottenere la nuova risorsa, ma perdere alcune di quelle esistenti a favore di processi concorrenti. Proporre un miglioramento di questo schema.
26. Uno studente di informatica a cui è stato assegnato un lavoro sui deadlock, ha un'idea brillante per eliminare i deadlock. Quando un processo richiede una risorsa, viene anche specificato un limite di tempo. Se il processo è bloccato perché la risorsa non è disponibile, viene avviato un timer; se il tempo limite è superato, il pro-

cesso viene rilasciato e gli si permette di essere eseguito di nuovo. Se foste il professore, quale voto dareste a questa proposta e perché?

27. Cenerentola e il Principe stanno divorziando. Per dividere le loro proprietà hanno stabilito il seguente algoritmo. Ogni mattina, ognuno di loro può spedire una lettera all'avvocato dell'altro per richiedere una parte delle proprietà. Poiché ci vuole una giornata perché le lettere arrivino a destinazione, essi si sono messi d'accordo che, nel caso scoprano di aver richiesto la stessa cosa nello stesso giorno, il giorno seguente manderanno una lettera per disdire la richiesta. Fra le loro proprietà ci sono un cane, Woofer, la sua cuccia, il loro canarino, Tweeter, e la sua gabbia. Poiché gli animali adorano le proprie casette, hanno deciso di non considerare valida nessuna spartizione dei beni che separi un animale dalla propria casetta, e che, qualora si presentasse una situazione di questo tipo, l'intero procedimento debba essere iniziato da capo. Sia Cenerentola che il Principe desiderano disperatamente Woofer. Per poter andare in vacanza (separatamente), hanno programmato due personal computer per gestire il negoziato. Quando ritornano dalla vacanza, i due calcolatori stanno ancora trattando. Perché? È possibile avere stalli? È possibile avere situazioni di attesa infinita? Discutere.
28. Uno studente che sta specializzandosi in antropologia ed è appassionato di calcolatori, si è fatto coinvolgere in un progetto di ricerca per studiare se i babbuini africani possono essere istruiti per riconoscere le situazioni di stalli. Lo studente localizza un canyon profondo e fissa una corda da una parte all'altra del canyon in modo che i babbuini possano attraversarlo appendendosi con le mani. Più babbuini possono attraversarlo contemporaneamente, supponendo che vadano tutti nella stessa direzione, ma se lo spostamento dei babbuini avvenisse contemporaneamente da est e da ovest, si avrebbe un deadlock sulla corda (i babbuini si fermerebbero a metà) perché i babbuini non riescono ad arrampicarsi l'uno sull'altro mentre sono sospesi alla corda in mezzo al canyon. Se un babbuino vuole attraversare il canyon deve controllare che non ci siano altri babbuini che in quel momento stiano attraversando nel senso opposto. Scrivere un programma che utilizzi dei semafori per evitare il deadlock, senza preoccuparsi del fatto che ci possa essere una serie di babbuini provenienti da est che impedisca per un tempo indefinito lo spostamento da ovest.
29. Ripetere il problema precedente tenendo conto ora della eventuale mancanza di risorse (starvation). Nel caso che un babbuino voglia andare ad est, se arrivando alla corda trova dei babbuini che la stanno usando per andare ad ovest, deve attendere che la corda sia vuota ma non è permesso ad altri babbuini diretti ad ovest di iniziare la traversata finché almeno un babbuino non è riuscito a traversare in senso opposto.
30. Programmare una simulazione dell'algoritmo del banchiere. Il programma dovrebbe eseguire un ciclo fra tutti i clienti della banca, valutando se le loro richieste siano sicure o meno. L'output deve essere un file delle richieste e delle decisioni.

GESTIONE DELLA MEMORIA

La memoria è una risorsa importante e va gestita con attenzione. Sebbene un personal computer dei nostri giorni abbia una quantità di memoria 1000 volte superiore a quella dell'IBM 7094 (il calcolatore più grande del mondo nei primi anni 60), i programmi stanno crescendo in dimensione a velocità maggiore di quella con cui crescono le memorie. Parafrasando la legge di Parkinson, possiamo dire che "I programmi si espandono fino a riempire la memoria disponibile". In questo capitolo studieremo come i sistemi operativi gestiscono la loro memoria.

Idealmente, quello che ogni programmatore vorrebbe avere è una memoria infinitamente grande ed infinitamente veloce, che sia anche non volatile, cioè, che non perda il suo contenuto quando viene tolta la corrente elettrica. Già che ci siamo, perché non chiediamo anche alla memoria di non essere costosa? Sfortunatamente la tecnologia non ci fornisce tali menorie, di conseguenza la maggior parte dei computer hanno una gerarchia di memoria, con una piccola quantità di cache di memoria molto veloce, dispendiosa e volatile, decine di megabyte di memoria principale volatile (RAM) a velocità media e a prezzo medio, e decine o centinaia di gigabyte di disco di memorizzazione non volatile, lento ed economico. È compito del sistema operativo coordinare come vengono usate queste memorie.

La parte di sistema operativo che gestisce la gerarchia di memoria si chiama gestore della memoria. Il suo compito è quello di tenere conto delle parti di memoria usate e di quelle inutilizzate, di allocare memoria ai processi quando ne abbiano bisogno e di deallocastrarla quando hanno finito, e infine di gestire gli scambi fra memoria principale e disco, quando la memoria principale non è abbastanza grande per contenere tutti i processi.

In questo capitolo, esamineremo un certo numero di schemi di gestione della memoria, a partire da schemi molto semplici per arrivare fino a quelli più sofisticati. Comince-

remo dall'inizio e vedremo per primo il sistema più semplice di gestione della memoria; poi, in maniera graduale, arriveremo a quelli sempre più elaborati.

Come abbiamo visto nel Capitolo 1, nel mondo dei computer la storia tende a ripetersi; gli schemi più semplici di gestione della memoria non vengono più utilizzati per i computer da tavolo, mentre sono ancora utilizzati in alcuni palmari, nei sistemi embedded, e nei sistemi smart card, per questo motivo, vale ancora la pena studiarli.

4.1 Sistemi di base per la gestione della memoria

I sistemi di gestione della memoria possono essere divisi in due classi: quelli che spostano i processi avanti e indietro dai dischi alla memoria durante l'esecuzione (sistemi con swapping o paginazione) e quelli che non lo fanno. Questi ultimi sono più semplici, così li studieremo per primi; più tardi, in questo stesso capitolo esamineremo lo swapping e la paginazione. Per tutto il capitolo, il lettore deve ricordarsi che sia lo swapping sia la paginazione sono marcheggi dettati per lo più dall'insufficienza della memoria principale che non è in grado di contenere tutti i programmi; se mai la memoria diventerà così grande da contenerli, le motivazioni in favore dell'adozione di uno schema di gestione della memoria o di un altro possono diventare obsolete.

D'altra parte, come detto sopra, il software sembra crescere molto più velocemente della memoria, quindi una gestione efficiente della memoria potrebbe sempre essere necessaria. Negli anni 80, c'erano molte università che utilizzavano un sistema a condivisione di tempo con dozzine di utenti (più o meno soddisfatti) su un sistema VAX a 4 MB; oggi la Microsoft raccomanda di avere almeno 64 MB per un sistema Windows 2000 a singolo utente. La tendenza ad avere sistemi multimediali porta ad avere ancora più richiesta in termini di memoria, quindi, per almeno i prossimi 10 anni, sarà probabilmente necessaria una buona gestione della memoria.

4.1.1 Monoprogrammazione senza swapping o paginazione

Lo schema di gestione della memoria più semplice possibile è quello di eseguire soltanto un programma alla volta, condividendo la memoria tra il programma ed il sistema operativo. Le tre variazioni a questo schema sono mostrate in Figura 4.1: il sistema operativo può stare nella parte bassa della memoria RAM (Random Access Memory, memoria ad accesso casuale), come in Figura 4.1(a), oppure può trovarsi in ROM (Read Only Memory, memoria a sola lettura) nella memoria alta, come in Figura 4.1(b), oppure i gestori dei dispositivi possono risiedere in ROM nella memoria alta e il resto del sistema operativo in RAM nella memoria bassa, come in Figura 4.1(c). Il primo modello era utilizzato principalmente sui mainframe e sui minicomputer, ma attualmente è scarsamente utilizzato; il secondo modello è utilizzato in alcuni computer palmari e in alcuni sistemi embedded; il terzo modello era utilizzato nei vecchi personal computer (ad esempio quelli che utilizzavano MS-DOS), dove la porzione del sistema nella ROM è chiamata BIOS (Basic Input Output System, sistema operativo elementare per l'ingresso/uscita).

Quando il sistema viene organizzato in questo modo, in un generico istante di tempo può essere in esecuzione un solo processo. Appena l'utente batte un comando dal terminale, il sistema operativo carica il programma richiesto dal disco in memoria e lo esegue. Quando il processo termina, il sistema operativo scrive un carattere di prompt sul terminale ed attende un comando dal terminale per caricare un nuovo processo, sovrapponendolo al primo.

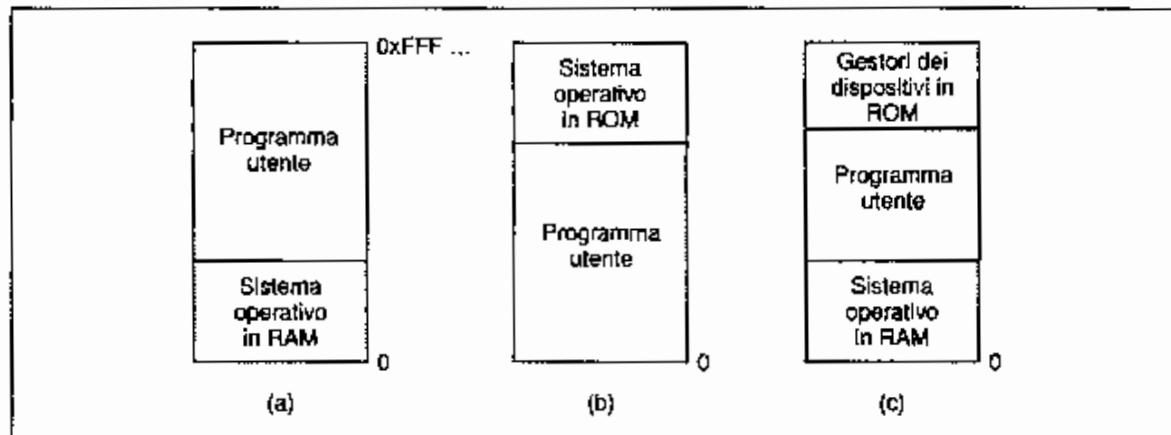


Figura 4.1 Tre semplici modi di organizzare la memoria con un sistema operativo ed un processo utente. Esistono anche altre possibilità.

4.1.2 Multiprogrammazione con partizioni fisse

Ad eccezione dei semplici sistemi embedded, la monoprogrammazione attualmente non viene più utilizzata; la maggior parte dei moderni sistemi operativi consentono a diversi processi di girare contemporaneamente. Avere diversi processi che girano contemporaneamente significa che quando un processo è bloccato in attesa di I/O, un altro processo può utilizzare la CPU, perciò la multiprogrammazione incrementa l'utilizzo della CPU. I server di rete hanno sempre la capacità di far girare contemporaneamente diversi processi (per diversi clienti), ma oggi giorno anche la maggior parte delle macchine client (per esempio desktop) hanno questa capacità.

Il modo più semplice per realizzare la multiprogrammazione è semplicemente quello di dividere la memoria in n partizioni, magari diverse. Questa suddivisione può essere fatta manualmente dagli operatori, per esempio, quando il sistema viene avviato; quando arriva un job, può essere messo nella coda d'ingresso della partizione più piccola che lo possa contenere. Dal momento che le partizioni sono fisse, in questo schema, tutto lo spazio di una partizione non usato dal job va sprecato. In Figura 4.2(a) vediamo come appare questo schema di partizioni fisse e di code d'ingresso di job separate.

Lo svantaggio dell'organizzare i job in ingresso in code separate diventa evidente nel caso in cui la coda per una partizione grande sia vuota ma quella per una partizione piccola sia piena, come nel caso delle partizioni 1 e 3 in Figura 4.2(a); qui piccoli job devono aspettare per essere inseriti in memoria, sebbene la maggior parte della memoria sia libera. Un'organizzazione alternativa è quella di mantenere una singola coda, come in Figura 4.2(b). Ogni qualvolta una partizione diventa libera, vi viene caricato il job più vicino alla testa della coda che può entrare nella partizione, ed esso è quindi mandato in esecuzione. Dal momento che non è desiderabile sprecare una partizione molto grande per far girare un job piccolo, un'altra strategia è quella di cercare in tutta la coda in ingresso ogni volta che si libera una partizione, e scegliere il job più grande che può entrarci. Da notare che quest'ultimo algoritmo discrimina negativamente i job piccoli, considerandoli immeritevoli di un'intera partizione, mentre di solito è preferibile dare ai job più piccoli (che spesso sono interattivi) il servizio migliore, non quello peggiore.

Una via d'uscita è di avere sempre a disposizione una partizione piccola, che permetterà ai job piccoli di girare senza dover allocare per loro una partizione grande.

Un altro approccio prevede una regola che dice che un job che può andare in esecuzione non venga lasciato da parte per più di k volte. Ogni volta che viene lasciato da par-

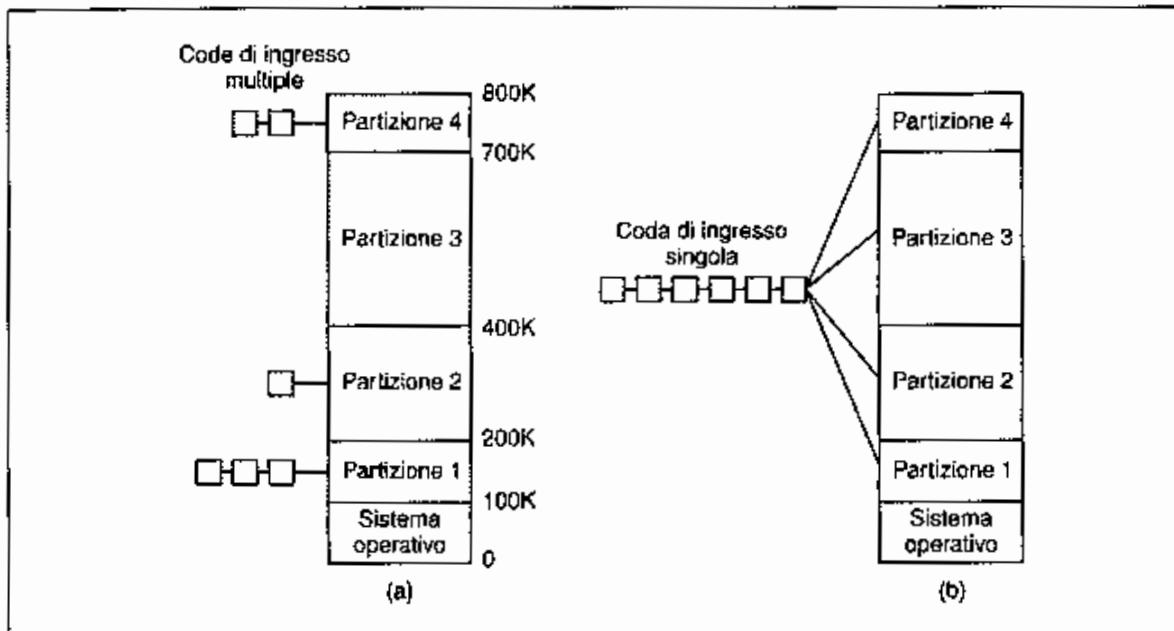


Figura 4.2 [a] Partizioni di memoria fisse con code d'ingresso separate per ciascuna partizione.
[b] Partizioni di memoria fisse con una singola coda d'ingresso.

te, prende un punto; quando è arrivato a k punti, non può essere lasciato da parte.

Questo sistema, con le partizioni fissate dall'operatore la mattina e non più cambiate, è stato usato dall'OS/360 sui grandi mainframe IBM e veniva chiamato MFT (Multiprogramming with a Fixed number of Task, multiprogrammazione con un numero fisso di task o OS/MFT). È semplice da capire ed egualmente semplice da implementare: i job in arrivo vengono accodati finché non è disponibile una partizione adatta, ed a quel punto un job viene caricato e mandato in esecuzione fino a che non termina. Oggigiorno supportano questo modello pochi sistemi operativi, forse nessuno.

4.1.3 Modello della multiprogrammazione

Usando la multiprogrammazione, si può incrementare l'utilizzo della CPU. In termini semplicistici, se un processo mediamente esegue computazioni per il 20% del tempo in cui si trova in memoria, con cinque processi contemporaneamente in memoria la CPU potrebbe essere occupata per tutto il tempo. Tuttavia, questo modello è irrealisticamente ottimistico, dal momento che suppone che tutti e cinque i processi non siano mai contemporaneamente in attesa dell'ingresso/uscita.

Un modello migliore consiste nel considerare la CPU da un punto di vista probabilistico. Supponiamo che un processo spenda una frazione p del suo tempo in attesa del completamento di operazioni di I/O: con n processi contemporaneamente in memoria, la probabilità che tutti gli n processi stiano aspettando il completamento di un'operazione di ingresso/uscita (nel qual caso la CPU sarebbe inattiva) è p^n . L'utilizzo della CPU è dato dalla formula

$$\text{utilizzo CPU} = 1 - p^n$$

La Figura 4.3 mostra l'utilizzo della CPU come funzione di n , che viene detto grado di multiprogrammazione.

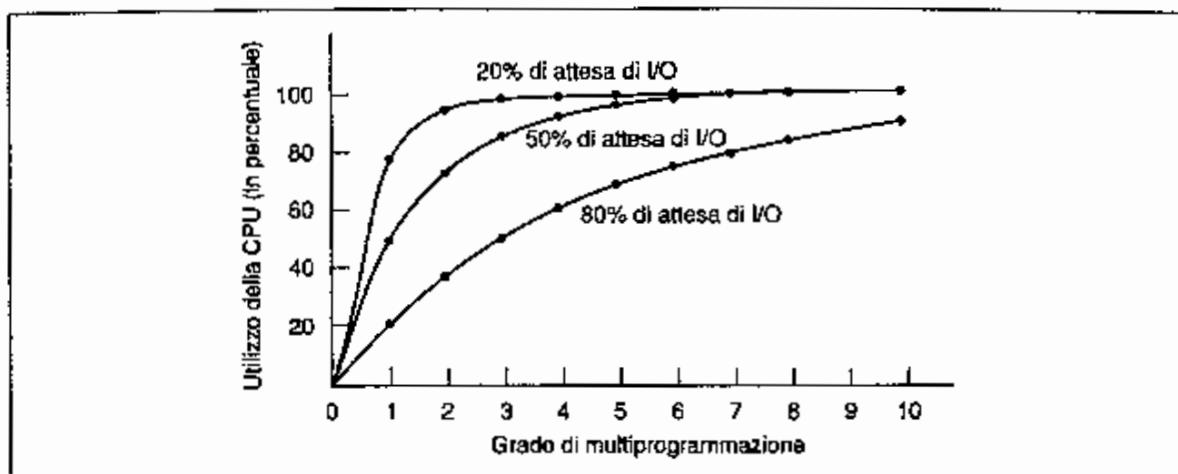


Figura 4.3 Utilizzo della CPU in funzione del numero dei processi in memoria.

Dalla figura risulta che se i processi spendono l'80% del proprio tempo in operazioni di ingresso/uscita, per mantenere lo spreco del tempo di CPU al disotto del 10% bisogna mantenere contemporaneamente in memoria almeno 10 processi. Se si pensa che ogni processo interattivo, in attesa che l'utente batte qualche cosa al terminale, è in uno stato di attesa di un'operazione di ingresso/uscita, vi risulterà chiaro che i tempi di attesa dell'80% e oltre non sono insoliti. Anche nei sistemi batch, i processi che fanno molte operazioni di ingresso/uscita dal disco o dal nastro avranno spesso percentuali analoghe o maggiori.

Per maggior precisione e completezza, bisogna sottolineare che il modello probabilistico appena descritto è solo un'approssimazione: esso suppone implicitamente che tutti gli n processi siano indipendenti, intendendo che è accettabile, per un sistema con cinque processi in memoria, averne tre in esecuzione e due sospesi. Ma con una sola CPU, non possiamo avere tre processi contemporaneamente in esecuzione, per cui un processo che diventa pronto mentre la CPU è occupata dovrà aspettare; quindi i processi non sono indipendenti. Un modello più accurato può essere costruito utilizzando la teoria delle code, ma l'osservazione appena fatta, cioè che la multiprogrammazione permette ai processi di utilizzare la CPU quando essa sarebbe altrimenti inattiva, è ovviamente ancora valida, anche se le curve di Figura 4.3 sono leggermente diverse.

Sebbene il modello di Figura 4.3 sia semplicistico, può ancora essere usato per fare previsioni specifiche, anche se approssimate, sulle prestazioni della CPU. Supponiamo, ad esempio, che un calcolatore abbia 32 MB di memoria, con un sistema operativo che occupi fino a 16 MB e con ciascun programma utente che occupi 4 MB: queste dimensioni permettono di avere quattro programmi utente contemporaneamente in memoria. Con un 80 per cento di attesa media per l'I/O, abbiamo un utilizzo della CPU (ignorando il lavoro del sistema operativo) del 60 per cento ($1 - 0,8^4$). L'aggiunta di altri 16 MB di memoria permette al sistema di passare da una multiprogrammazione a quattro ad una multiprogrammazione a otto processi, portando la percentuale di utilizzo della CPU all'83 per cento. In altre parole, i 16 MB addizionali di memoria incrementano il throughput (rendimento) del 38 per cento.

L'aggiunta di altri 16 MB porterebbe l'utilizzo della CPU solo dall'83 al 93 per cento, migliorando il throughput solo di un altro 12 per cento. Usando questo modello il proprietario del calcolatore potrebbe decidere che la prima aggiunta sia un buon investimento, ma che la seconda non lo sia.

4.1.4 Analisi delle prestazioni dei sistemi a multiprogrammazione

Il modello descritto sopra può essere usato anche per analizzare i sistemi di tipo batch. Consideriamo, per esempio, un centro di calcolo i cui programmi in media passano un 80 per cento del loro tempo in attività di ingresso/uscita; una certa mattina si presentano per l'esecuzione quattro job, come mostrato in Figura 4.4(a). Il primo job, che arriva alle 10 di mattina, richiede 4 minuti di CPU; con l'80 per cento di tempo speso in operazioni di ingresso/uscita, il job usa solo 12 secondi di CPU per ogni minuto in cui si trova in memoria, anche se non ci sono altri job in competizione per la CPU. Gli altri 48 secondi vengono spesi aspettando che le operazioni di ingresso/uscita siano completate. In questo modo, il job deve rimanere in memoria per almeno 20 minuti in modo da riuscire a svolgere i 4 minuti di attività di CPU, anche in assenza di competizione nell'uso della stessa CPU.

Dalle 10 alle 10 e 10, il job 1 ha la memoria tutta per sé e riesce a lavorare per due minuti. Quando arriva il job numero 2, alle 10 e 10, l'utilizzo della CPU cresce da 0,20 a 0,36, a causa del più alto grado di multiprogrammazione (vedi Figura 4.3); tuttavia, con uno scheduler round robin, ciascun job ottiene metà CPU, così riesce a lavorare per 0,18 minuti per ogni minuto che rimane in memoria. Si noti che l'aggiunta di un secondo job costa al primo solo un 10 per cento delle sue prestazioni (da 0,20 a 0,18 minuti di CPU per minuto di tempo reale).

Alle 10 e 15 arriva un terzo job; a questo punto, il job numero 1 ha ottenuto 2,9 minuti di CPU e il job numero 2 ha avuto 0,9 minuti di CPU. Con la multiprogrammazione di grado 3, ogni job riceve 0,16 minuti di CPU per minuto, come mostrato in Figura 4.4(b). Dalle 10 e 15 alle 10 e 20, ciascuno dei tre job ottiene 0,8 minuti di tempo di CPU; alle 10 e 20 arriva un quarto job. La Figura 4.4(c) mostra la sequenza di eventi completa.

4.1.5 Rilocazione e protezione

La multiprogrammazione introduce due problemi essenziali che devono essere risolti: la rilocazione e la protezione. Dalla Figura 4.2 risulta chiaro che job diversi vanno in esecuzione ad indirizzi diversi. Quando viene effettuato il link di un programma (il programma principale, le procedure scritte dall'utente e le procedure di libreria vengono combinati in un singolo spazio di indirizzi), il linker deve conoscere l'indirizzo di partenza del programma in memoria.

Per esempio, supponiamo che la prima istruzione sia una chiamata alla procedura che si trova all'indirizzo assoluto 100 nel file binario prodotto dal linker. Se il programma viene caricato nella partizione 1 (all'indirizzo 100K), quell'istruzione salterà all'indirizzo assoluto 100, che sta dentro al sistema operativo. Ciò che è necessario è una chiamata all'indirizzo 100K + 100. Se il programma viene caricato nella partizione 2, deve essere eseguito come una chiamata alla locazione 200K + 100, e così via. Questo problema è noto come il problema della **rilocazione**.

Una possibile soluzione è la modifica delle istruzioni quando il programma viene caricato in memoria. I programmi caricati nella partizione 1 aggiungeranno 100K ad ogni indirizzo, i programmi caricati nella partizione 2 aggiungeranno 200K agli indirizzi, e così via. Per effettuare in questo modo la rilocazione durante il caricamento, il linker deve includere nel programma binario una lista, o una mappa di bit (bitmap) che dica quali parole del programma sono indirizzi da rilocare e quali sono codici di operazione, costanti o altre voci che non debbono essere rilocate. OS/MFT lavorava in questo modo.

La rilocazione durante il caricamento non risolve il problema della protezione; un programma malvagio può sempre costruire una nuova istruzione e saltarvi. Poiché i pro-

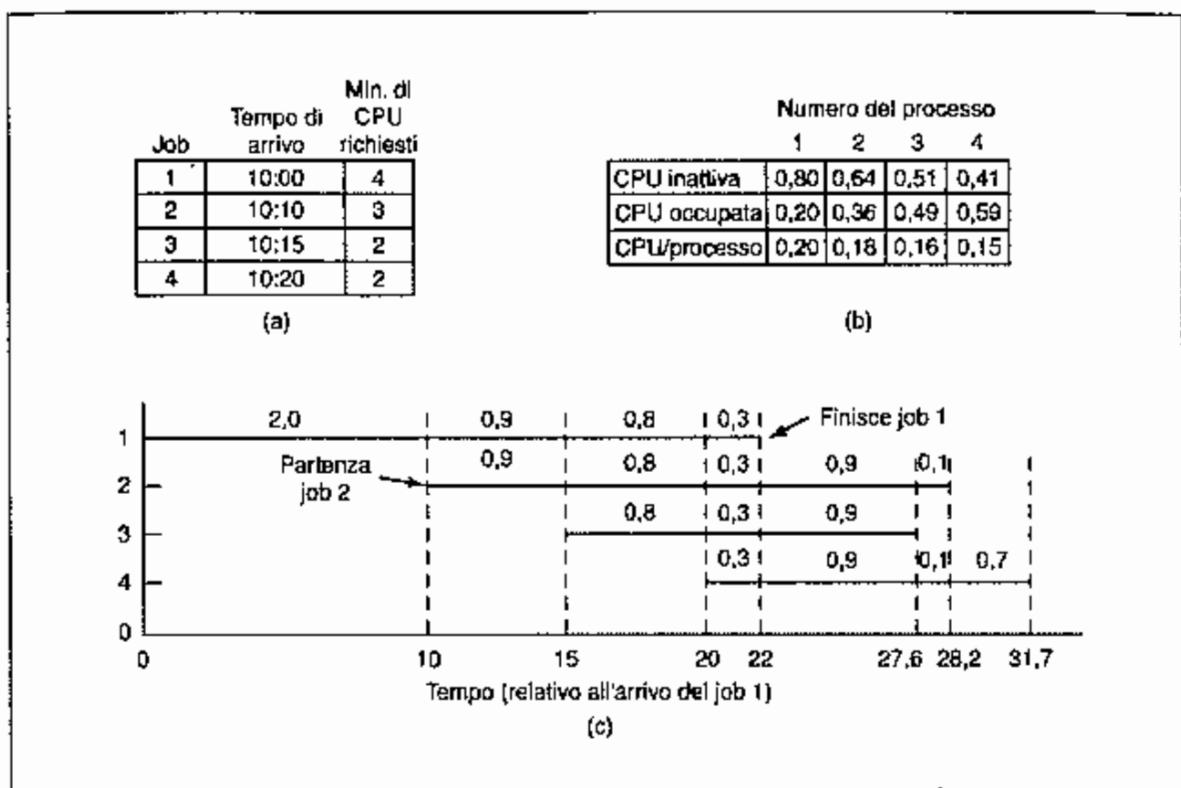


Figura 4.4 [a] Tempo di arrivo ed esigenze di quattro job. [b] Utilizzo della CPU con job che spendono l'80 per cento del loro tempo in attesa del completamento dell'ingresso/uscita da 1 a 4 job. [c] Sequenza di eventi man mano che i job arrivano e terminano. Il numero sopra alle righe orizzontali indica quanto tempo di CPU, in minuti, ottiene ciascun job per ciascun intervallo.

grammi in questo sistema usano indirizzi assoluti, invece che indirizzi relativi ad un registro, non c'è nessun modo per impedire ad un programma di costruire un'istruzione che legga o scriva qualunque parola in memoria. Nei sistemi multiutente non si vuole permettere ai processi di leggere e scrivere la memoria appartenente ad altri utenti.

La soluzione che scelse l'IBM per proteggere il 360 fu di dividere la memoria in blocchi di 2KB e di assegnare ad ogni blocco un codice di protezione a 4 bit. La PSW conteneva una chiave a 4 bit; l'hardware del 360 individuava qualunque tentativo da parte di un processo in esecuzione di accedere alla memoria il cui codice di protezione differiva dalla chiave contenuta nella PSW. Poiché soltanto il sistema operativo poteva cambiare i codici di protezione e la chiave, ai processi utente veniva impedito di interferire fra di loro e con il sistema operativo stesso.

Una soluzione alternativa ad entrambi i problemi della rilocazione e della protezione, è quella di dotare la macchina di due registri hardware speciali, chiamati **registro base** e **limite**. Quando un processo viene schedulato, nel registro base viene caricato l'indirizzo di inizio della sua partizione, e nel registro limite viene caricata la lunghezza della partizione. Ad ogni indirizzo di memoria generato viene automaticamente sommato il contenuto del registro base, prima di inviarlo alla memoria; quindi, se il registro base è 100K, un'istruzione CALL 100 viene in realtà convertita in una istruzione CALL 100K + 100, senza modificare l'istruzione stessa. Gli indirizzi vengono anche confrontati con il registro limite, per assicurarsi che non tentino di indirizzare al di fuori della partizione corrente; l'hardware protegge i registri base e limite per impedire ai programmi utente di modificarli.

Uno svantaggio di questo schema è dato dalla necessità di eseguire una somma ed un confronto ad ogni riferimento in memoria. I confronti possono essere eseguiti velocemente, ma le somme sono lente a causa del tempo di propagazione del riporto, a meno che non vengano utilizzati circuiti speciali per le somme.

Il CDC 6600, il primo supercomputer al mondo, utilizzava questo schema. La CPU dell'Intel 8088, utilizzata per il PC IBM originale, utilizzava una versione più debole di questo schema, con registri base ma senza registri limite, tuttavia pochi computer lo usano ancora.

4.2 Swapping

Con i sistemi batch, organizzare la memoria in partizioni fisse è facile ed efficiente. Ogni job, quando raggiunge la testa della coda, viene caricato in una partizione e rimane in memoria finché non ha terminato la sua esecuzione. Fino a quando si possono tenere in memoria abbastanza job per tenere sempre occupata la CPU, non c'è ragione di andare a cercare qualcosa di più complicato.

Con i sistemi timesharing (a condivisione di tempo) o con i personal computer orientati alla grafica, la situazione è diversa; a volte non vi è abbastanza memoria principale per mantenere tutti i processi correntemente attivi, così i processi in eccesso devono essere mantenuti su disco ed essere introdotti in memoria per essere eseguiti dinamicamente.

Possono essere utilizzati due approcci generali di gestione della memoria, a seconda (in parte) dell'hardware disponibile. La strategia più semplice, chiamata swapping, consiste nel caricare interamente in memoria ogni processo, eseguirlo per un tot di tempo e quindi spostarlo nuovamente su disco. L'altra strategia, chiamata memoria virtuale, consente ai programmi di girare anche quando sono caricati solo parzialmente nella memoria principale. Di seguito studieremo lo swapping; nella Sezione 4.3 esamineremo la memoria virtuale.

Le operazioni di un sistema con swapping vengono illustrate in Figura 4.5: inizialmente il solo processo A risiede in memoria, poi vengono creati (o caricati dal disco) i pro-

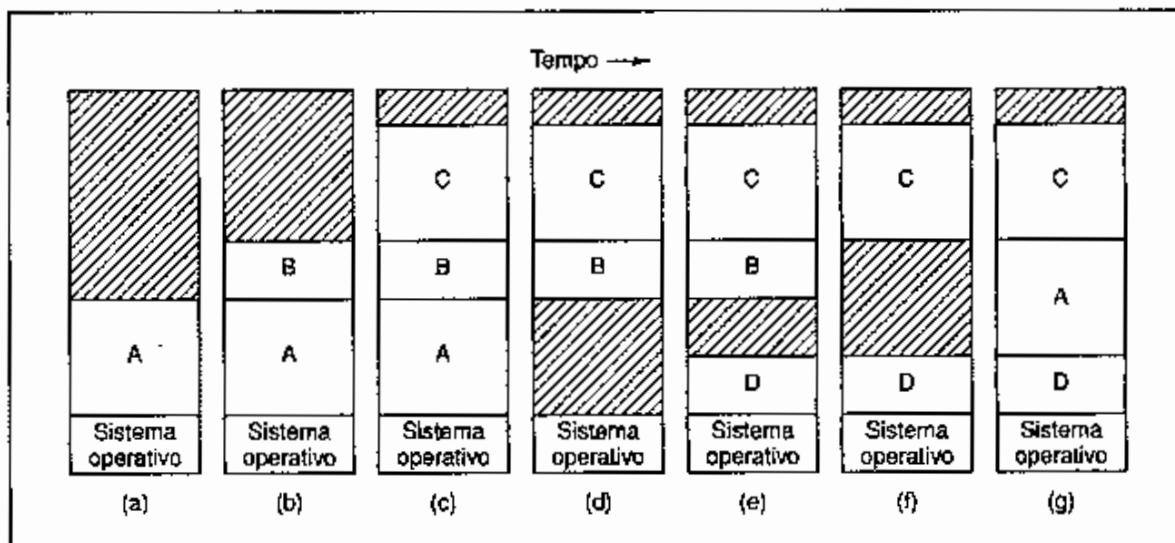


Figura 4.5 L'allocazione della memoria cambia man mano che i processi arrivano in memoria o la lasciano. Le regioni tratteggiate rappresentano la memoria non usata.

cessi B e C. In Figura 4.5(d) il processo A viene scaricato sul disco, poi arriva D, e B viene scaricato; infine, ritorna il processo A. Poiché ora A è inserito in una locazione diversa, gli indirizzi contenuti in esso devono essere rilocati, dal software nel momento in cui viene caricato oppure (più facilmente) dall'hardware durante l'esecuzione del programma.

La differenza principale fra le partizioni fisse della Figura 4.2 e le partizioni variabili della Figura 4.5 è che il numero, la dimensione e la posizione delle partizioni variano dinamicamente, nel secondo caso, con l'arrivo e la partenza dei processi, mentre nel primo caso sono fisse. La flessibilità data dal non essere legati ad un numero di partizioni che possono essere troppo grandi o troppo piccole migliora l'utilizzo della memoria, ma complica anche l'allocazione ed il rilascio della memoria, così come il tener traccia dell'allocazione.

Quando lo swapping crea molti buchi in memoria, è possibile combinarli in un unico, grande buco, spostando tutti i processi il più indietro possibile: questa tecnica è nota con il nome di **compattazione della memoria** e normalmente non viene eseguita, dal momento che richiede un sacco di tempo di CPU. Per esempio, su una macchina da 256 Mbyte che può copiare 4 byte ogni 40 nanosecondi, occorrono circa 2,7 secondi per compattare tutta la memoria.

Una cosa su cui vale la pena di riflettere è quanta memoria deve essere allocata quando si crea un processo o lo si carica in memoria. Se i processi vengono creati con una dimensione fissa, che non cambia mai, allora l'allocazione è semplice: si alloca esattamente la quantità di memoria necessaria, né più né meno.

Se tuttavia i segmenti dati dei processi possono crescere, per esempio allocando dinamicamente la memoria da uno heap, come accade in molti linguaggi di programmazione, allora si possono avere dei problemi quando un processo tenta di crescere. Se vicino al processo si trova un buco, esso può essere allocato al processo, ed il processo vi può

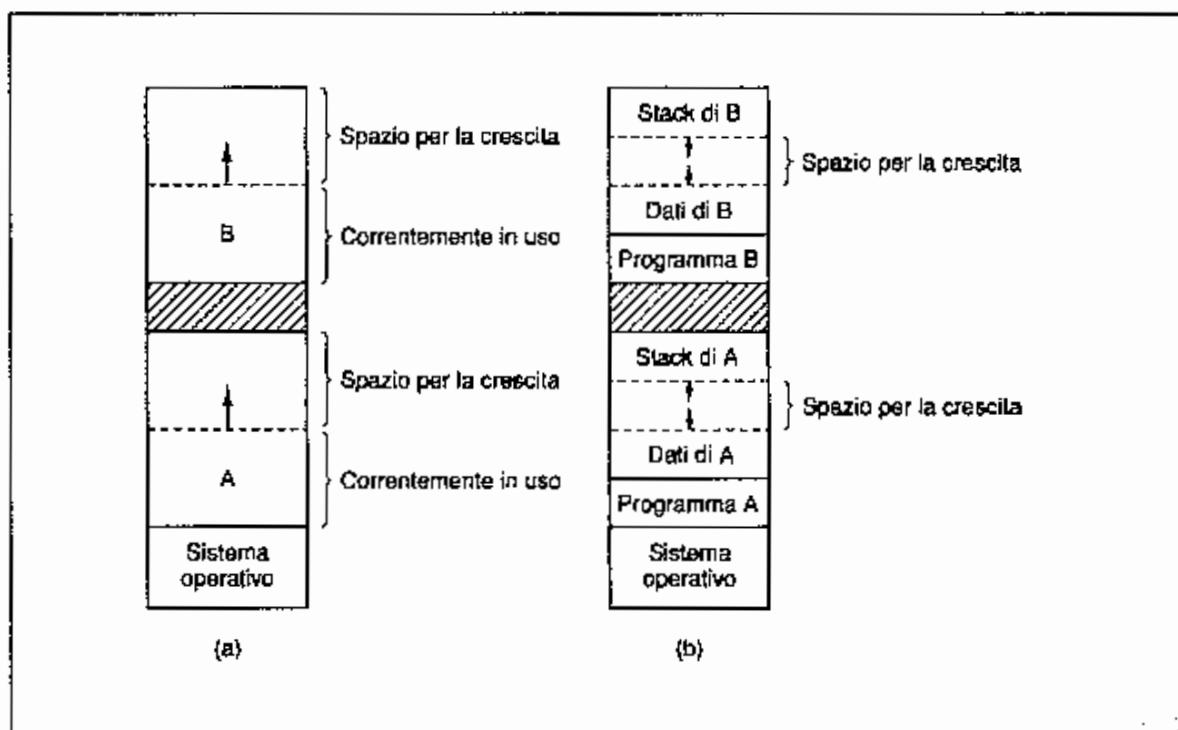


Figura 4.6 [a] Allocazione di uno spazio per un segmento di dati che cresce. [b] Allocazione di spazio per uno stack che cresce ed un segmento dati che cresce.

crescere. D'altra parte, se il processo è adiacente ad un altro processo, il processo in crescita deve essere spostato in un buco di memoria abbastanza grande per contenerlo, oppure occorre scaricare uno o più processi per poter creare un buco abbastanza grande per contenerlo. Se un processo non può crescere mentre è in memoria e l'area di swap (scambio) sul disco è piena, il processo deve attendere oppure deve essere ucciso.

Se ci si aspetta che la maggior parte dei processi cresca durante l'esecuzione, allora probabilmente è una buona idea allocare una piccola quantità di memoria extra quando un processo viene caricato dal disco o spostato in memoria, per ridurre l'overhead dovuto allo scaricamento o allo spostamento quando un processo non entra più nella memoria che gli è stata allocata. Ad ogni buon conto, quando un processo viene scaricato sul disco, va scaricata solo la memoria che viene effettivamente utilizzata; salvare anche la memoria extra allocata al processo è uno spreco. In Figura 4.6(a) vediamo una configurazione di memoria nella quale è stato allocato spazio per la crescita a due processi.

Se i processi avessero due segmenti che crescono, per esempio, il segmento di dati, utilizzato come heap per le variabili che sono allocate e rilasciate dinamicamente, ed il segmento stack per le variabili locali normali e gli indirizzi di ritorno, è evidente che si potrebbe utilizzare uno schema alternativo come quello di Figura 4.6(b). In questa figura vediamo che ciascun processo ha uno stack che cresce verso il basso, posto in cima alla memoria che gli è stata assegnata, e un segmento di dati posto proprio sopra il testo del programma, che cresce verso l'alto. La memoria che si trova fra i due segmenti può essere usata dall'uno o dall'altro; se si esaurisce, il processo deve essere spostato in un buco abbastanza grande, oppure scaricato sul disco finché non si crea un buco abbastanza grande, oppure deve essere ucciso.

4.2.1 Gestione della memoria con mappe di bit

Quando la memoria viene assegnata dinamicamente, il sistema operativo deve gestirla. In generale, ci sono due modi per tenere traccia dell'utilizzo della memoria: le mappe di bit e le liste libere. In questa sezione e nella successiva analizzeremo questi due metodi uno dopo l'altro.

Con una mappa di bit, la memoria viene divisa in unità di allocazione che possono essere lunghe solo poche parole o arrivare a qualche kilobyte; a ciascuna delle unità di allocazione viene associato un bit della mappa, che vale 0 se l'unità è libera e 1 se è occupata (o viceversa). La Figura 4.7 mostra una parte di memoria e la corrispondente mappa di bit.

Le dimensioni delle unità di allocazione rappresentano una scelta di progetto importante. Più è piccola l'unità di allocazione e più è grande la mappa di bit; tuttavia, anche con unità di allocazione di soli 4 byte, 32 bit di memoria richiederanno solamente 1 bit nella mappa di bit. Una memoria di $32n$ bit userà n bit nella mappa di bit, così la mappa di bit prenderà solo $1/33$ della memoria. Se si sceglie un'unità di allocazione grande, allora la mappa di bit è piccola, ma viene sprecata una quantità significativa di memoria nell'ultima unità quando la dimensione dei processi non è un multiplo esatto dell'unità di allocazione.

Una mappa di bit mette a disposizione una maniera semplice per tener conto dello stato delle parole di memoria usando una quantità di memoria fissa; infatti, la dimensione della mappa di bit dipende solo dalla dimensione della memoria fisica e dalla dimensione dell'unità di allocazione. Il problema principale sta nel fatto che quando si decide di caricare in memoria un processo di k unità, il gestore della memoria deve esaminare la mappa di bit per trovare una sequenza consecutiva di k bit a 0. La ricerca di una sequen-

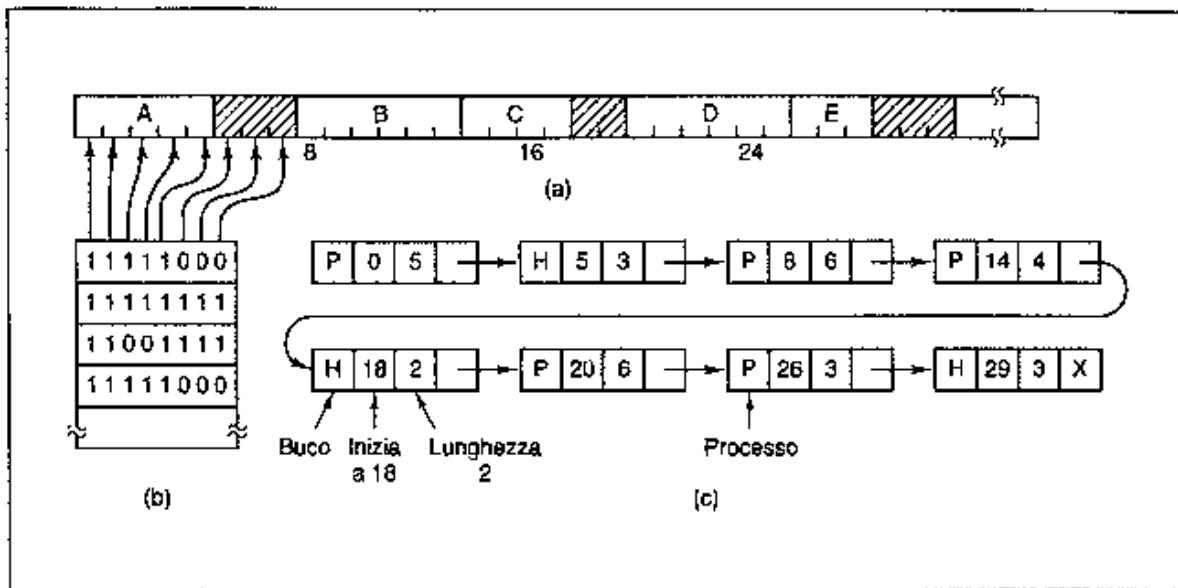


Figura 4.7 [a] Una parte della memoria con cinque processi e tre buchi (hole). I separatori segnano le unità di allocazione. Le regioni tratteggiate (0 nella mappa di bit) sono libere. [b] La mappa di bit corrispondente. [c] La stessa informazione rappresentata come lista concatenata.

za di bit in una mappa di bit è un'operazione lenta poiché occorre controllare se la sequenza si trova a cavallo di due parole della mappa di bit; questo è un problema tipico delle mappe di bit.

4.2.2 Gestione della memoria con liste concatenate

Un altro modo per tener traccia dello stato della memoria è di mantenere una lista concatenata dei segmenti di memoria liberi e occupati, dove un segmento è un processo oppure un buco fra due processi; la memoria di Figura 4.7(a) è rappresentata in Figura 4.7(c) come lista concatenata di segmenti. Ciascun elemento nella lista specifica un buco (H) o un processo (P), l'indirizzo di partenza, la lunghezza ed un puntatore al prossimo elemento.

In questo esempio, la lista dei segmenti è ordinata per indirizzo. Ordinarla in questo modo presenta il vantaggio che quando un processo termina o viene scaricato l'aggiornamento della lista è molto semplice. Un processo che termina ha normalmente due vicini (a meno che non sia il processo all'estremo superiore o inferiore della memoria); questi possono essere processi o buchi, il che porta alle quattro configurazioni della Figura 4.8. Nella Figura 4.8(a) l'aggiornamento della lista comporta la sostituzione di un elemento P con uno H. Nella Figura 4.8(b) e 4.8(c), due elementi vengono fusi in uno solo e la lista diventa più corta di un elemento. In Figura 4.8(d) vengono raggruppati tre elementi, e dalla lista ne vengono rimossi due. Dal momento che l'elemento della tabella dei processi relativo al processo che termina contiene il puntatore all'elemento della lista che contiene tale processo, risulta più conveniente mantenere una lista a doppia concatenazione, anziché a singola come in Figura 4.7(c). Questa struttura rende più semplice la ricerca dell'elemento precedente per vedere se è possibile eseguire una fusione fra elementi.

Quando i processi e i buchi vengono mantenuti in una lista concatenata ordinata per indirizzo, si possono usare diversi algoritmi per allocare la memoria ad un processo

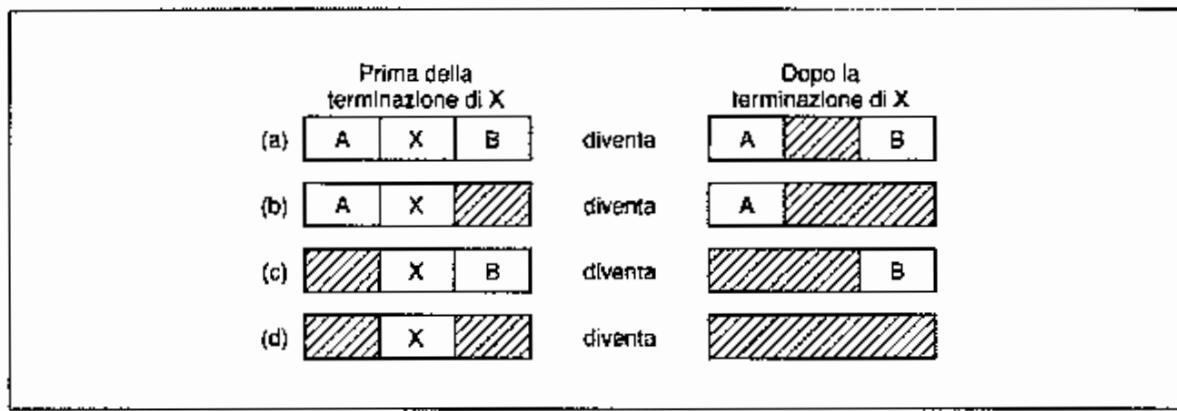


Figura 4.B Quattro combinazioni di vicini per il processo X che termina.

appena creato o ricaricato dal disco. Supponiamo che il gestore della memoria sappia quanta memoria allocare: l'algoritmo più semplice è detto **first fit** (primo posto sufficiente). Il gestore della memoria scorre la lista di segmenti fino a che non trova un buco che sia abbastanza grande; il buco viene quindi diviso in pezzi, uno per il processo ed uno per la memoria che non viene usata, tranne che nel caso abbastanza improbabile in cui il buco sia esattamente della lunghezza cercata. L'algoritmo first fit è un algoritmo molto veloce perché esegue la ricerca più corta possibile.

Una variante del first fit è il **next fit** (prossimo posto sufficiente). Lavora nella stessa maniera del first fit, tranne per il fatto che si ricorda di dove ha trovato un buco adatto; la prossima volta che viene chiamato inizia la ricerca da dove si era fermato, anziché cominciare sempre dall'inizio, come fa l'algoritmo first fit. Simulazioni condotte da Bays (1977) dimostrano che il next fit ha delle prestazioni leggermente peggiori del first fit.

Un altro algoritmo molto conosciuto è il **best fit** (miglior posto sufficiente), che cerca in tutta la lista e prende il più piccolo dei buchi che possono essere usati. Anziché spezzare un buco grosso che potrebbe servire più tardi, il best fit cerca di trovare un buco le cui dimensioni siano molto vicine a quelle effettivamente necessarie.

Come esempio di first fit e best fit, considerate di nuovo la Figura 4.7. Se occorre un blocco di lunghezza 2, il first fit allocherà il buco che si trova alla posizione 5, mentre il best fit allocherà il buco alla posizione 18.

Il best fit è più lento del first fit dal momento che deve scorrere tutta la lista ogni volta che viene chiamato; sorprendentemente però, dà anche luogo ad un maggiore spreco di memoria rispetto al first fit o al next fit, dal momento che tende a riempire la memoria di buchi piccoli ed inutili. Il first fit genera in media buchi più grossi.

Per evitare che dopo la divisione di un buco per collocarvi il processo la parte rimanente sia di dimensioni troppo piccole, si potrebbe pensare di adottare un algoritmo **worst fit** (peggiore posto sufficiente) e cioè, di prendere sempre il buco più grande disponibile, cosicché il buco generato possa avere dimensione abbastanza grande da risultare utile. Tramite simulazioni, si è visto che anche il worst fit non è una buona idea.

Tutti e quattro gli algoritmi possono essere resi più veloci mantenendo liste separate per buchi e processi; in questo modo, tutti gli algoritmi dedicano le loro energie a cercare fra i buchi e non fra i processi. Il prezzo pagato per accelerare l'allocazione è la complessità aggiuntiva ed il rallentamento che si hanno quando un blocco di memoria viene deallocated, dal momento che il segmento rilasciato deve essere rimosso dalla lista dei processi ed inserito in quella dei buchi.

Se si mantengono liste distinte per processi e buchi, la lista dei buchi può essere ordinata per dimensione, per rendere più veloce il best fit. Quando il best fit esegue la ricer-

ca di un buco partendo da quelli di dimensioni più piccole fino a quelli di dimensioni più grandi, non appena trova un buco sufficiente sa che quello è il buco di dimensioni più piccole che fa allo scopo, e quindi quello "best fit". Non è necessaria nessuna ricerca successiva, come invece accade nello schema ad una lista singola. Con la lista dei buchi ordinata per dimensioni, best fit e first fit sono ugualmente veloci, mentre next fit non ha alcun senso.

Quando i buchi vengono tenuti in liste separate da quelle dei processi, si può effettuare una piccola ottimizzazione. Invece di avere un insieme separato di strutture dati per mantenere la lista dei buchi, come in Figura 4.7(c), si possono usare gli stessi buchi: la prima parola di ogni buco può essere usata per la dimensione del buco e la seconda per il puntatore al prossimo elemento. I nodi della lista di Figura 4.7(c), che richiedono tre parole più un bit (P/H) ciascuno, non sono quindi più necessari.

Un altro algoritmo di allocazione è il quick fit, che mantiene liste separate per alcune delle dimensioni richieste più di frequente. Per esempio, potrebbe avere una tabella con n elementi, nella quale il primo elemento sia un puntatore alla testa della lista dei buchi da 4K, il secondo elemento un puntatore alla lista dei buchi da 8K, il terzo elemento un puntatore alla lista dei buchi da 12K e così via. Buchi di 21K, per esempio, potrebbero essere messi nella lista dei buchi da 20K o in una speciale lista di buchi di dimensioni diverse. Con il quick fit, trovare un buco delle dimensioni richieste risulta estremamente veloce, ma ha lo stesso svantaggio di tutti gli schemi che ordinano i buchi per dimensione e cioè, risulta dispendioso trovare i vicini di un processo che termina o che viene scaricato, per vedere se è possibile fondere dei blocchi. Se la fusione non avvenisse, la memoria si frammenterebbe rapidamente in un gran numero di buchi piccoli ed inutili.

4.3 Memoria virtuale

Molti anni fa, la gente dovette confrontarsi per la prima volta con programmi che erano troppo grandi per poter essere caricati in memoria centrale; la soluzione generalmente adottata era quella di dividere il programma in parti, dette overlay. L'overlay 0 cominciava a girare per primo, e quando aveva finito chiamava un altro overlay. Alcuni sistemi ad overlay erano molto complessi, e permettevano la coesistenza in memoria di più overlay alla volta. Gli overlay venivano mantenuti sul disco e caricati nella memoria dal sistema operativo, dinamicamente, quando richiesto.

Sebbene il lavoro necessario al caricamento e scaricamento venisse in realtà svolto dal sistema operativo, il compito di dividere il programma in parti era del programmatore. Dividere i grossi programmi in piccole parti modulari era un'attività che portava via tempo e molto noiosa. Non ci volle molto prima che qualcuno pensasse ad un modo per delegare al calcolatore l'intero lavoro.

Il metodo che fu trovato (Fotheringham, 1961) divenne noto con il nome di memoria virtuale. L'idea di base che sta dietro alla memoria virtuale è che la dimensione combinata di programma, dati e stack può eccedere la dimensione della memoria fisica per essi disponibile. Il sistema operativo mantiene in memoria le parti che sono in uso in un certo momento, mentre le altre parti vengono mantenute sul disco. Per esempio, un programma da 16-MB può girare in una macchina da 4 MB scegliendo in maniera accurata quali 4 MB mantenere in memoria ad ogni istante, con pezzi di programma che vengono caricati e scaricati fra disco e memoria secondo le necessità.

La memoria virtuale può anche funzionare con sistemi multiprogrammati, con pezzi e bocconi di molti programmi in memoria allo stesso tempo. Quando un programma sta

attendendo che parti di esso vengano caricate in memoria, è in attesa di operazioni di I/O e non può girare, così la CPU può passare ad eseguire un altro processo, allo stesso modo che in ogni altro sistema a multiprogrammazione.

4.3.1 La paginazione

La maggior parte dei sistemi con memoria virtuale usa una tecnica, detta **paginazione**, che ora descriveremo. Su un qualunque calcolatore, esiste un insieme di indirizzi di memoria che possono essere prodotti dai programmi. Quando un programma usa un'istruzione tipo

```
MOV REG, 1000
```

esso sposta il contenuto della memoria all'indirizzo 1000 dentro a REG (o viceversa, a seconda della macchina). Gli indirizzi possono essere generati usando l'indicizzazione, i registri base, i registri segmento ed altri metodi.

Questi indirizzi generati dal programma vengono detti **indirizzi virtuali** e formano lo **spazio di indirizzamento virtuale**. Sui calcolatori privi di memoria virtuale, l'indirizzo virtuale viene direttamente spedito sul bus della memoria e causa la lettura o la scrittura della parola di memoria fisica che ha quello stesso indirizzo. Quando si usa la memoria virtuale, gli indirizzi virtuali non finiscono direttamente sul bus della memoria. Invece, vanno a finire ad una **unità della gestione della memoria** (MMU, Memory Management Unit), che inappa gli indirizzi virtuali sugli indirizzi della memoria fisica come illustrato in Figura 4.9.

Un esempio molto semplice del funzionamento di questa trasformazione viene mostrato in Figura 4.10. In questo esempio, abbiamo un calcolatore che può generare indirizzi da 16 bit, da 0 a 64K; questi sono indirizzi virtuali. Questo calcolatore, tuttavia, ha solo 32KB di memoria fisica, cosicché, anche se si possono scrivere programmi da 64KB, questi non

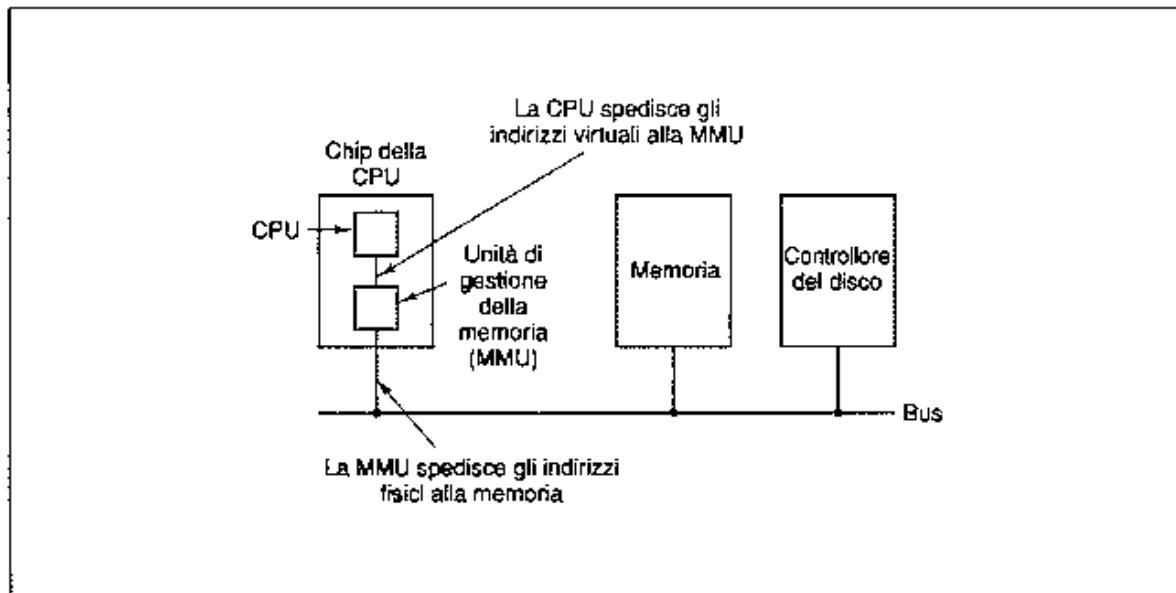


Figura 4.9 La posizione e il funzionamento della MMU. Qui la MMU è mostrata come parte del chip della CPU poiché al giorno d'oggi comunemente sta in questo chip; naturalmente potrebbe essere un chip separato così com'era anni fa.

possono essere caricati interamente in memoria e fatti girare. Così occorre mantenere una copia dell'intera immagine del programma su disco (fino ad un massimo di 64KB) in maniera tale che se ne possano prendere dei pezzi secondo necessità.

Lo spazio di indirizzamento virtuale viene diviso in unità chiamate **pagine**: le corrispondenti unità nella memoria fisica vengono dette **pagine fisiche** (page frame). Le pagine e le pagine fisiche hanno sempre la stessa dimensione; in questo esempio sono di 4KB, ma comunemente si usano pagine che vanno da 512 byte a 64KB. Con 64KB di spazio di indirizzamento virtuale e 32KB di memoria fisica abbiamo 16 pagine virtuali e 8 pagine fisiche. I trasferimenti da e per il disco vengono sempre effettuati in unità di una pagina.

Quando un programma tenta di accedere all'indirizzo 0, per esempio, con l'istruzione

```
MOV REG, 0
```

l'indirizzo virtuale 0 viene mandato alla MMU. La MMU vede che l'indirizzo cade nella pagina 0 (da 0 a 4095) la quale, secondo la sua funzione di traduzione corrisponde alla pagina fisica 2 (da 8192 a 12287); di conseguenza trasforma l'indirizzo in 8192, e manda in uscita l'indirizzo 8192 sul bus. La scheda di memoria non sa niente della MMU, vede semplicemente apparire una richiesta per leggere o scrivere l'indirizzo 8192 e di conseguenza, la onora. Quindi la MMU ha effettivamente trasformato tutti gli indirizzi virtuali fra 0 e 4095 in indirizzi fisici fra 8192 e 12287.

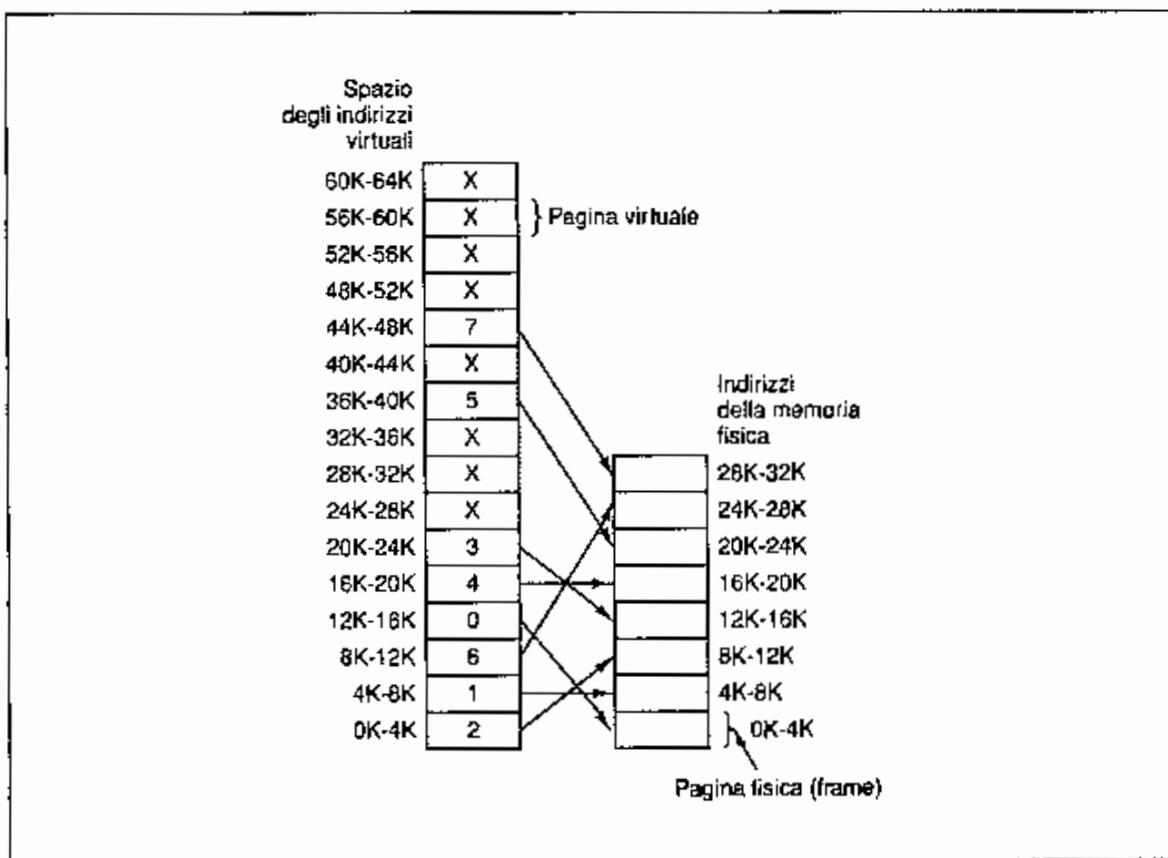


Figura 4.10 La relazione fra indirizzi virtuali e indirizzi di memoria fisica è data dalla tabella delle pagine.

Analogamente, una istruzione

`MOV REG, 8192`

viene in realtà trasformata in una istruzione

`MOV REG, 24576`

dal momento che l'indirizzo virtuale 8192 sta nella pagina 2 e questa pagina è mappata sulla pagina fisica 6 (indirizzi fisici da 24576 a 28671). Come terzo esempio, l'indirizzo virtuale 20500 sta a 20 posizioni dall'inizio della pagina virtuale 5 (indirizzi virtuali da 20480 a 24575) e si trasforma nell'indirizzo fisico $12288 + 20 = 12308$.

Da sola, questa capacità di mappare 16 pagine virtuali su una qualunque delle otto pagine fisiche definendo opportunamente la funzione di traduzione della MMU non risolve il problema che deriva dal fatto che lo spazio di indirizzamento virtuale è più grosso della memoria fisica. Dal momento che abbiamo solamente otto pagine fisiche, solo otto delle pagine virtuali della Figura 4.10 vengono mappate sulla memoria fisica; le altre, segnate con una croce nella figura, non sono mappate. Nell'hardware vero, un bit **presente/assente** memorizza se la pagina sia mappata o meno.

Cosa succede se un programma tenta di usare una pagina non mappata, per esempio tramite l'istruzione

`MOV REG, 32780`

che accede il dodicesimo byte della pagina virtuale 8 (che comincia all'indirizzo 32768)? La MMU si accorge che la pagina non è mappata (indicata con una croce in figura) e fa in modo che la CPU esegua una trap al sistema operativo; questo passaggio viene chiamato **fault di pagina** (page fault). Il sistema operativo prende una delle pagine fisiche poco usate e ne scrive il contenuto sul disco; successivamente preleva la pagina appena riferita e la mette nella pagina fisica resa libera, cambia la funzione di traduzione e fa ripartire l'istruzione che era stata interrotta.

Per esempio, se il sistema operativo decidesse di rimuovere la pagina 1, caricerebbe la pagina virtuale 8 all'indirizzo 4K e farebbe due cambiamenti nella funzione di traduzione della MMU. Per prima cosa, contrassegnerebbe l'elemento della pagina 1 come non mappato, per bloccare ogni successivo tentativo di accesso agli indirizzi virtuali compresi fra 4K e 8K; successivamente, sostituirebbe la crocetta nell'elemento relativo alla pagina 8 con un 1, così che quando l'istruzione interrotta verrà rieseguita trasformerà l'indirizzo virtuale 32780 nell'indirizzo fisico 4108.

Adesso diamo un'occhiata dentro alla MMU per vedere come funziona e perché abbiamo scelto di usare pagine che abbiano una dimensione pari ad una potenza di 2. In Figura 4.11 vediamo un esempio di indirizzo virtuale, 8196 (001000000000100 in binario), che viene tradotto usando la mappa della MMU di Figura 4.10. L'indirizzo virtuale in ingresso di 16 bit viene diviso in un numero di pagina da 4 bit e in un offset (spiazzamento) da 12 bit: con i 4 bit del numero di pagina possiamo rappresentare 16 pagine e con i 12 bit dell'offset possiamo indirizzare tutti i 4096 byte che appartengono ad una pagina.

Il numero di pagina viene usato come indice nella **tabella delle pagine**, che contiene il numero della pagina fisica corrispondente alla pagina virtuale. Se il bit **presente/assente** è a 0, si provoca una trap al sistema operativo; se il bit è a 1, il numero di pagina fisica trovato nella tabella delle pagine viene copiato nei 3 bit di ordine più alto del registro di uscita, insieme ai 12 bit dell'offset, che vengono copiati senza modifiche

dall'indirizzo virtuale. Insieme, essi formano un indirizzo fisico di 15 bit; il registro di uscita viene quindi posto sul bus della memoria come indirizzo di memoria fisica.

4.3.2 Tabelle delle pagine

In teoria, la traduzione degli indirizzi virtuali in indirizzi fisici funziona come abbiamo appena descritto. L'indirizzo virtuale viene diviso in un numero di pagina virtuale (i bit di ordine superiore) e in un offset (i bit di ordine inferiore). Per esempio, con un indirizzo a 16 bit e una pagina di dimensione 4 KB, i 4 bit più alti potrebbero specificare una delle 16 pagine virtuali, e i 12 bit più bassi dovrebbero quindi specificare l'offset in byte (da 0 a 4095) entro la pagina selezionata. Tuttavia, è ugualmente possibile prendere tre, cinque o qualche altro numero di bit per indirizzare la pagina: differenti suddivisioni implicano differenti dimensioni della pagina.

Il numero di pagina virtuale viene usato come indice nella tabella delle pagine per trovare l'elemento relativo a quella pagina virtuale; da quest'ultimo si deriva il numero di pagina fisica (se esiste); questo viene attaccato alla parte alta dell'offset, rimpiazzando il numero di pagina virtuale, per formare un indirizzo fisico che può essere spedito alla memoria.

Lo scopo della tabella delle pagine è quello di mappare pagine virtuali in pagine fisiche. Da un punto di vista matematico, la tabella delle pagine è una funzione, con il numero di pagina virtuale come argomento e il numero di pagina fisica come risultato. Usando il risultato di questa funzione, il campo che denota la pagina virtuale in un indirizzo

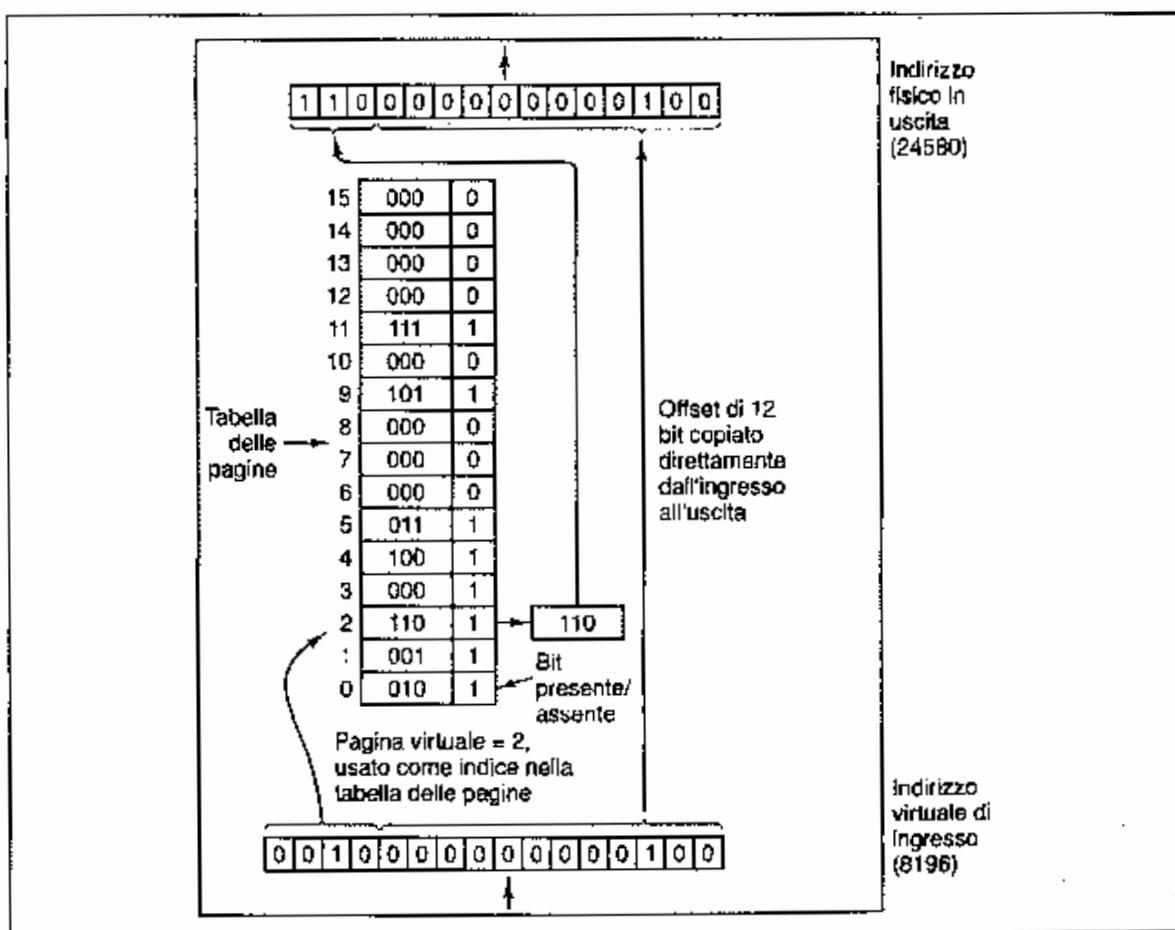


Figura 4.11 Il funzionamento interno della MMU con 16 pagine da 4K.

virtuale può essere sostituito dal numero della pagina fisica, in modo da formare un indirizzo di memoria fisica.

Nonostante questa descrizione semplice, occorre affrontare due problemi principali:

1. La tabella delle pagine può essere molto grande.
2. La traduzione deve essere veloce.

Il primo punto deriva dal fatto che i calcolatori moderni usano indirizzi virtuali di almeno 32 bit; con pagine di 4KB, per esempio, uno spazio di indirizzamento da 32 bit ha un milione di pagine e uno spazio da 64 bit ne ha più di quante ne possiate immaginare. Con un milione di pagine nello spazio di indirizzamento virtuale, la tabella delle pagine dovrebbe avere un milione di elementi, e ricordate che ogni processo ha bisogno di una sua tabella delle pagine (poiché ha il suo personale spazio di indirizzamento virtuale).

Il secondo punto è una conseguenza del fatto che la traduzione da indirizzi virtuali a indirizzi fisici deve essere eseguita per ogni riferimento alla memoria. Una istruzione tipica occupa una parola di memoria, e spesso ha anche un operando in memoria; di conseguenza, è necessario eseguire 1,2 ed in qualche caso anche più di 2 riferimenti alla tabella delle pagine per ciascuna istruzione. Se una istruzione dura, per esempio, 4 ns, la ricerca nella tabella deve essere eseguita in meno di 1 nanosecondo per evitare che diventi un evidente collo di bottiglia.

La necessità di una tabella delle pagine ampia e veloce rappresenta un vincolo significativo nel progetto dei calcolatori. Sebbene il problema sia più serio nelle macchine ad alte prestazioni, rappresenta un problema anche nelle macchine della fascia bassa, dove risulta critico il costo ed il rapporto prezzo/prestazioni. In questa sezione e nelle prossime vedremo in dettaglio il progetto delle tabelle delle pagine e mostreremo un certo numero di soluzioni hardware che vengono usate nei calcolatori veri.

Lo schema più semplice (almeno dal punto di vista concettuale) consiste nell'avere una sola tabella delle pagine realizzata tramite un vettore di registri hardware molto veloci, con un elemento per ognuna delle pagine virtuali indirizzato con il numero di pagina virtuale, come mostrato in Figura 4.11. Quando viene fatto partire un processo, il sistema operativo carica i registri con la tabella delle pagine del processo, presa da una copia tenuta nella memoria principale; durante l'esecuzione del processo, non sono più necessari altri riferimenti alla memoria relativi alla tabella delle pagine. I vantaggi di questo metodo consistono nel fatto che è semplice e che non richiede riferimenti alla memoria durante la traduzione; uno svantaggio sta nel fatto che può risultare molto dispendioso (se la tabella delle pagine è grande): il fatto stesso di dover caricare la tabella delle pagine ad ogni cambio di contesto può degradare le prestazioni.

All'altro estremo, la tabella delle pagine può risiedere interamente in memoria. Tutto ciò di cui ha bisogno l'hardware è un solo registro che punti all'inizio della tabella in memoria principale. Questo schema permette di cambiare la tabella delle pagine ad ogni cambio di contesto tramite il caricamento di un solo registro; naturalmente ha lo svantaggio di richiedere uno o più riferimenti alla memoria per leggere la tabella delle pagine durante l'esecuzione di ogni istruzione. Per questa ragione, questo approccio viene usato raramente nella sua versione più pura, ma di seguito studieremo delle variazioni che presentano prestazioni molto migliori.

• Tabelle delle pagine a più livelli

Per risolvere il problema derivante dalla presenza costante in memoria di tabelle delle pagine molto grosse, molti calcolatori usano una tabella delle pagine a più livelli. Un

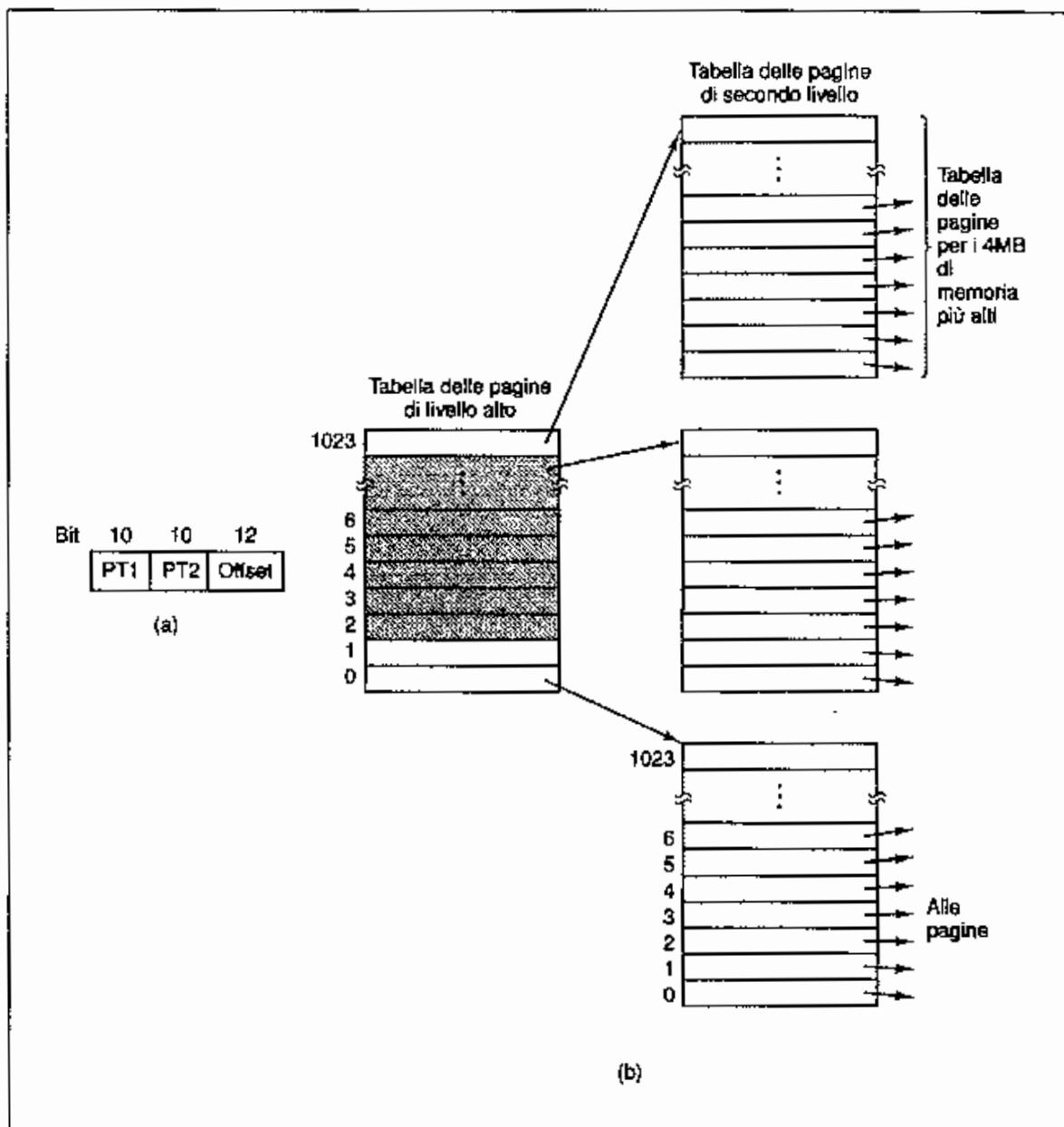


Figura 4.12 (a) Un indirizzo da 32 bit con due campi di pagina. (b) Una tabella delle pagine a due livelli.

semplice esempio è mostrato in Figura 4.12. In Figura 4.12(a) abbiamo un indirizzo virtuale da 32 bit che viene diviso in un campo PT1 da 10 bit, un campo PT2 da 10 bit e un campo Offset da 12 bit. Dal momento che gli offset sono da 12 bit, le pagine sono da 4KB e ce ne sono un totale di 2^{20} .

Il segreto del metodo delle tabelle a più livelli è di evitare di tenere sempre tutte le tabelle delle pagine in memoria. In particolare, quelle che non risultano necessarie non devono essere tenute in giro. Supponiamo, per esempio, che un processo necessiti di 12 MB: i 4 MB più bassi per il programma, i 4 MB successivi per i dati, e gli ultimi 4 MB per la pila. Fra la cima dei dati e il fondo dello stack c'è un gigantesco buco che non viene usato.

In Figura 4.12(b) vediamo come funziona una tabella delle pagine a due livelli in questo esempio. Sulla sinistra abbiamo la tabella delle pagine di più alto livello, con 1024 ele-

menti che corrispondono ai 10 bit del campo *PT1*. Quando si presenta un indirizzo virtuale alla MMU, essa dapprima estrae il campo *PT1* e lo usa come indice per accedere alla tabella delle pagine di più alto livello. Ciascuno dei 1024 elementi di questa tabella rappresenta 4M dal momento che l'intero spazio di indirizzamento virtuale da 4 gigabyte (corrispondente ai 32 bit) è stato spezzettato in 1024 parti.

L'elemento individuato accedendo alla tabella delle pagine di più alto livello dà il numero della pagina fisica del secondo livello della tabella delle pagine. L'elemento 0 della tabella di più alto livello punta alla tabella delle pagine del programma, l'elemento 1 punta alla tabella delle pagine dei dati e l'elemento 1023 punta a quella dello stack. Gli altri elementi (ombreggiati) non vengono usati. Il campo *PT2* viene ora usato come indice nella tabella delle pagine di secondo livello appena selezionata, per trovare il numero della pagina fisica corrispondente alla pagina virtuale.

Come esempio, considerate l'indirizzo virtuale a 32 bit 0x00403004 (4 206 596, in decimale) che corrisponde al byte 12292 dello spazio dei dati. Questo indirizzo ha *PT1* = 1, *PT2* = 3 e *Offset* = 4. La MMU usa prima *PT1* per indirizzare la tabella delle pagine di più alto livello ed ottenere l'elemento 1, che corrisponde agli indirizzi da 4M a 8M. Poi usa *PT2* per indirizzare la tabella delle pagine di secondo livello appena trovata, e per estrarre l'elemento 3, che corrisponde agli indirizzi da 12288 a 16383 nel proprio blocco da 4M (corrispondente agli indirizzi assoluti da 4 206 592 a 4 210 687). Questo elemento contiene il numero di pagina fisica della pagina che contiene l'indirizzo virtuale 0x00403004; se la pagina non è in memoria, il bit *presente/assente* nell'elemento della tabella delle pagine sarà zero, il che provocherà un fault di pagina. Se la pagina è in memoria, il numero di pagina fisica preso dalla tabella delle pagine di secondo livello viene combinato con l'offset (4) per costruire un indirizzo fisico che viene posto sul bus e spedito alla memoria.

La cosa interessante da notare sulla Figura 4.12 è che sebbene lo spazio di indirizzamento contenga più di un milione di pagine, in realtà sono necessarie solo quattro pagine: la tabella di più alto livello e le tabelle di secondo livello per gli indirizzi da 0 a 4M, da 4M a 8M e per gli ultimi 4M. Il bit *presente/assente* nei 1021 elementi della tabella delle pagine di più alto livello sono messi a 0, in modo da forzare un fault di pagina in caso di accesso. Se questo succede, il sistema operativo noterà che il processo tenta di fare riferimento ad una zona di memoria non lecita ed eseguirà l'azione opportuna, come per esempio spedirgli un segnale o ucciderlo. In questo esempio abbiamo scelto numeri semplici per le varie dimensioni e abbiamo preso *PT1* uguale a *PT2*, ma nella pratica si possono naturalmente usare altri valori.

Il sistema di tabelle di pagine a due livelli della Figura 4.12 può essere espanso a tre, quattro o più livelli; i livelli aggiuntivi danno maggiore flessibilità ma è dubbio che valga la pena di avere più complessità per andare oltre i tre livelli.

4 Struttura degli elementi della tabella delle pagine

Lasciamo ora i principi generali dell'organizzazione della struttura delle tabelle delle pagine per concentrarci sui dettagli relativi all'organizzazione del singolo elemento della tabella. L'esatta organizzazione di un elemento dipende in maniera pesante dalla macchina, ma il tipo di informazioni presenti è pressappoco lo stesso su tutte le macchine. Nella Figura 4.13 diamo un esempio di elemento della tabella delle pagine. Le dimensioni variano da calcolatore a calcolatore, ma di solito si usano 32 bit. Il campo più importante è il numero di pagina fisica; dopo tutto, lo scopo ultimo della tabella delle pagine è quello di trovare questo numero. Poi abbiamo il bit *presente/assente*. Se questo è 1, l'elemento è un elemento valido e può essere usato; se vale 0, la pagina virtuale cui corri-

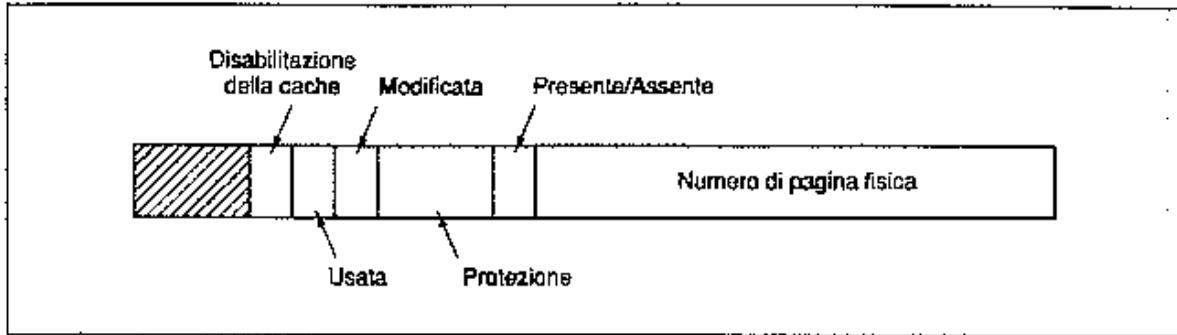


Figura 4.13 Un tipico elemento della tabella delle pagine.

sponde l'elemento non è attualmente presente in memoria. L'accesso ad un elemento della tabella delle pagine con questo bit a 0 provoca un fault di pagina.

I bit *protezione* ci dicono quali tipi di accesso sono permessi. Nella sua forma più semplice, questo campo è da 1 bit, con 0 che indica il diritto di lettura/scrittura e 1 che indica il diritto di sola lettura. Una soluzione più raffinata prevede l'uso di 3 bit, ciascuno dei quali abilita o disabilita la lettura, la scrittura e l'esecuzione della pagina.

I bit *modificata* e *usata* mantengono traccia dell'uso della pagina. Quando si scrive in una pagina, l'hardware mette automaticamente a 1 il bit *modificata*: questo serve al sistema operativo quando decide di reclamare una pagina fisica. Se la pagina è stata modificata (cioè è "sporca"), deve essere di nuovo scritta sul disco; se non è stata modificata (cioè è "pulita"), può essere semplicemente abbandonata, dal momento che la copia sul disco è ancora valida. Il bit, a volte, viene chiamato bit *sporco*, poiché riflette lo stato della pagina.

Il bit *usata* viene messo a 1 ogni volta che si fa un riferimento alla pagina, sia per la lettura che per la scrittura; serve ad aiutare il sistema operativo a scegliere una pagina da scaricare quando si verifica un fault di pagina. Le pagine che non sono state usate sono candidati migliori rispetto alle pagine usate e questo bit gioca un ruolo importante per diversi algoritmi di rimpiazzamento delle pagine che studieremo più avanti in questo stesso capitolo.

Infine, l'ultimo bit permette di disabilitare il caching della pagina; questa caratteristica è importante per le pagine che vengono mappate su registri di dispositivi anziché sulla memoria. Se il sistema operativo sta eseguendo un loop molto stretto aspettando che qualche dispositivo di ingresso/uscita risponda ad un comando che gli è appena stato dato, è essenziale che l'hardware continui a leggere la locazione dal dispositivo anziché usare la copia (vecchia) contenuta nella cache; tramite questo bit si può escludere il caching. Questo bit non è necessario sulle macchine che hanno uno spazio di ingresso/uscita separato e che non usano ingresso/uscita mappato in memoria.

Si noti che l'indirizzo del disco usato per mantenere la pagina quando non si trova in memoria non fa parte della tabella delle pagine. La ragione è semplice: la tabella delle pagine mantiene solo quelle informazioni che servono all'hardware per tradurre indirizzi virtuali in indirizzi fisici. Le informazioni necessarie al sistema operativo per trattare i fault di pagina vengono mantenute in tabelle che stanno dentro il sistema operativo, di cui l'hardware non ha bisogno.

4.3.3 TLB: Translation Lookaside Buffer

Nella maggior parte degli schemi di paginazione, le tabelle delle pagine vengono mantenute in memoria, a causa della loro grande dimensione, e questo ha un'enorme

impatto sulle prestazioni. Consideriamo, per esempio, una istruzione che copi un registro in un altro registro: in assenza di paginazione, l'istruzione esegue un solo riferimento alla memoria, dovuto al prelievo dell'istruzione stessa. Con la paginazione, sono necessari ulteriori riferimenti alla memoria per accedere alla tabella delle pagine; dal momento che la velocità di esecuzione è limitata, in generale, dalla velocità con cui la CPU può prelevare istruzioni e dati dalla memoria, produrre due riferimenti della tabella delle pagine per ogni riferimento in memoria, riduce le prestazioni di 2/3. A queste condizioni, nessuno vorrebbe usare la paginazione.

I progettisti di calcolatori conoscono questo problema da anni, ed alla fine hanno trovato una soluzione che si basa su questa osservazione: la maggior parte dei programmi tende ad eseguire un alto numero di riferimenti ad un piccolo insieme di pagine, e non viceversa. Così solo una piccola frazione degli elementi della tabella delle pagine viene utilizzata frequentemente, il resto non viene usato quasi mai.

La soluzione pensata consiste nel dotare i calcolatori di un piccolo dispositivo hardware che serve a mappare gli indirizzi virtuali sugli indirizzi fisici senza passare dalla tabella delle pagine. Il dispositivo, chiamato TLB (Translation Lookaside Buffer) (o a volte memoria associativa), è illustrato in Figura 4.14. Di solito si trova dentro alla MMU e contiene un piccolo numero di elementi, otto in questo esempio, di solito non più di 64: ciascun elemento contiene informazioni che riguardano una pagina, ed in particolare contiene il numero della pagina virtuale, un bit che viene messo a 1 quando la pagina viene modificata, il codice di protezione (permessi di lettura/scrittura/esecuzione), e la pagina fisica in cui si trova la pagina virtuale. Questi campi sono in corrispondenza uno a uno con i campi della tabella delle pagine; un altro bit indica se l'elemento è valido (cioè attualmente in uso) oppure no.

Un esempio che può generare il TLB di Figura 4.14 è un processo in un ciclo che attraversa le pagine virtuali 19, 20 e 21, di modo che queste pagine sono presenti nel TLB, con i bit di protezione che ne permettono la lettura e l'esecuzione. I dati principali correntemente in uso (per esempio una matrice da elaborare) stanno nelle pagine 129 e 130, e la pagina 140 contiene gli indici usati nei calcoli sulla matrice. Infine, lo stack si trova nelle pagine 860 e 861.

Vediamo ora come funziona il TLB. Quando alla MMU arriva un indirizzo virtuale da tradurre, l'hardware controlla dapprima se il relativo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente (cioè in parallelo) con tutti gli elementi. Se si trova un elemento con lo stesso numero di pagina virtuale (page hit) e non si ha violazione dei bit di protezione, il numero della pagina fisica viene direttamente preso dal TLB, senza bisogno di dover accedere alla tabella delle pagine. Se il numero di pagina vir-

Elemento valido	Pagina virtuale	Modificata	Protezione	Pagina fisica
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 4.14 Un TLB usato per accelerare la paginazione.

tuale risulta presente nel TLB ma l'istruzione sta tentando di scrivere in una pagina a sola lettura, viene generato un errore di protezione, in maniera del tutto analoga a quanto verrebbe fatto in seguito ad un accesso alla tabella delle pagine.

Il caso interessante si ha quando il numero di pagina virtuale non è presente nel TLB. La MMU si accorge della mancanza (page miss) ed esegue una normale ricerca nella tabella delle pagine; in seguito, scarica uno degli elementi del TLB e lo rimpiazza con l'elemento della tabella delle pagine appena trovato. Così, se a quella pagina si farà riferimento nell'immediato futuro, si avrà un page hit, piuttosto che un page miss. Quando si cancella un elemento dal TLB, il bit di modifica viene copiato nella tabella delle pagine in memoria, mentre gli altri valori si trovano già nell'elemento che sta in memoria; quando si carica il TLB dalla tabella delle pagine, tutti i campi vengono presi dalla memoria.

Software per la gestione del TLB

Fino ad ora, abbiamo ipotizzato che ogni macchina che presenta una memoria virtuale paginata ha delle tabelle delle pagine riconosciute dall'hardware, e in più un TLB. Con questi presupposti, la gestione del TLB e il trattamento dei fault del TLB sono effettuati interamente dall'hardware della MMU; solo quando una pagina non è presente in memoria si verificano delle trap al sistema operativo.

Nel passato, questa ipotesi era vera, tuttavia, molte macchine moderne RISC, tra cui le SPARC, MIPS, Alpha e HP PA, gestiscono le pagine quasi completamente via software. Su queste macchine, gli elementi del TLB sono caricati esplicitamente dal sistema operativo; quando avviene una page miss nel TLB, anziché lasciare che la MMU cerchi nella tabella delle pagine e prelevi il riferimento alla pagina richiesto, genera semplicemente un fault di TLB e rimanda il problema al sistema operativo. Il sistema deve trovare la pagina, rimuovere un elemento dal TLB, inserirne uno nuovo, e fare ripartire l'istruzione che era fallita, e certamente, tutto questo deve essere fatto con una manciata di istruzioni poiché le page miss nel TLB accadono molto più frequentemente che i fault di pagina.

Sorprendentemente, se il TLB è ragionevolmente grande (ad esempio 64 elementi) in modo da ridurre il tasso di miss, la gestione via software del TLB risulta essere sufficientemente efficiente. Il guadagno principale è quello di avere una MMU più semplice, che libera un considerevole ammontare di spazio nei chip della CPU per la cache e per altre caratteristiche che possono migliorare le prestazioni. Il software per la gestione del TLB viene discusso da Uhlig ed altri (1994).

Sono state sviluppate diverse strategie per migliorare le prestazioni su macchine che gestiscono via software il TLB. Un approccio riguarda sia la riduzione delle miss di TLB sia la riduzione del costo di una miss di TLB nel momento in cui si verifica (Bala ed altri, 1994). Per ridurre le miss del TLB, qualche volta il sistema operativo può utilizzare la sua intuizione per determinare quali pagine verranno probabilmente utilizzate in seguito e per caricare in anticipo gli elementi per queste pagine nel TLB. Per esempio, quando un processo client spedisce un messaggio ad un processo server sulla stessa macchina, è molto probabile che il server sia attivato entro breve tempo. Conoscendo questo, mentre il sistema esegue la send, può anche controllare dove sono le code del server, i dati, e le pagine di stack, e quindi mapparli prima che possano causare un fault di TLB.

Il modo normale per gestire una miss di TLB, sia nell'hardware che nel software, è quello di andare nella tabella delle pagine e di eseguire operazioni di indicizzazione per localizzare la pagina riferita. Il problema nell'effettuare questa ricerca tramite software sta nel fatto che le pagine che contengono la tabella delle pagine possono non essere presenti nel TLB, e questo causerà altri fallimenti di TLB durante la gestione della miss. Questi fallimenti possono essere ridotti, mantenendo una grande cache software (per esem-

pio 4KB) di elementi di TLB in una locazione fissata la cui pagina è sempre mantenuta nel TLB: controllando inizialmente la cache software, il sistema operativo può sostanzialmente ridurre le miss del TLB.

4.3.4 Tabelle delle pagine invertite

Le tabelle delle pagine tradizionali del tipo descritto precedentemente richiedono un elemento per ogni pagina virtuale, in quanto sono indicizzate da un numero di pagina virtuale. Se lo spazio di indirizzamento è di 2^{32} byte, con 4096 byte per pagina, allora sono necessari più di 1 milione di elementi nella tabella delle pagine; come limite minimo, la tabella delle pagine dovrà essere almeno di 4 megabyte; su grossi sistemi, questa dimensione è realizzabile.

Tuttavia, visto che i computer da 64 bit diventano sempre più comuni, la situazione cambia drasticamente. Se lo spazio di indirizzamento è ora di 2^{64} byte, con pagine di 4KB, abbiamo bisogno di una tabella delle pagine di 2^{52} elementi, e con 8 byte per ciascun elemento, la tabella risulterà essere di oltre 30 milioni di gigabyte; logicamente utilizzare 30 milioni di gigabyte solo per la tabella delle pagine non è pensabile, né ora e nemmeno per i prossimi anni a venire, di conseguenza, è necessaria una soluzione differente per spazi di indirizzamento virtuali paginati di 64 bit.

Una soluzione è la **tabella delle pagine invertite**: in questa tabella, è presente un elemento per ogni pagina fisica nella memoria reale, invece che un elemento per ogni pagina nello spazio di indirizzamento virtuale. Per esempio, con indirizzi virtuali di 64 bit, una pagina di 4 KB, e 256 MB di RAM, una tabella delle pagine invertite richiede solo 65536 elementi, e questi elementi tengono traccia di quale (processo, pagina virtuale) è localizzato nella pagina fisica.

Sebbene le tabelle delle pagine invertite recuperino una gran quantità di spazio, almeno quando lo spazio di indirizzamento virtuale è più grande della memoria fisica, esse presentano un problema serio: la traduzione da virtuale a fisico diventa molto più difficile. Quando il processo *n* riferisce la pagina virtuale *p*, l'hardware non può più trovare

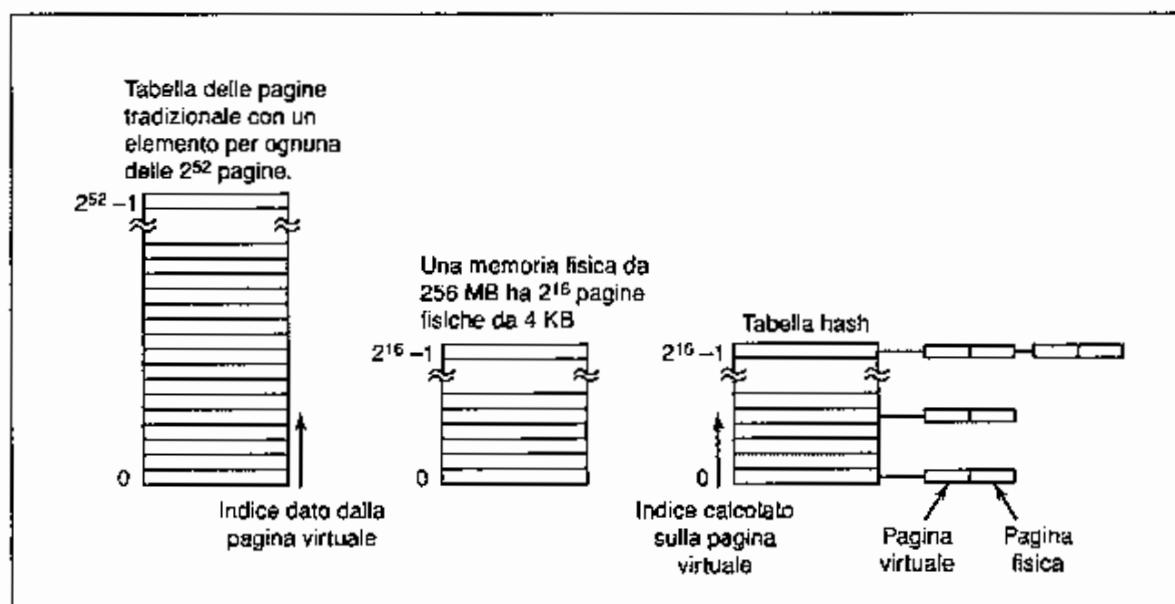


Figura 4.15 Confronto tra una tabella delle pagine tradizionale ed una tabella delle pagine invertite.

la pagina fisica utilizzando p come indice nella tabella delle pagine, dovrà invece ricercare nell'intera tabella delle pagine invertite un elemento del tipo (n, p) ; inoltre, questa ricerca deve essere fatta non solo sui fault di pagina, ma per ogni riferimento di memoria, e scorrere una tabella di 64K per ogni riferimento di memoria non è proprio il modo migliore per rendere la vostra macchina super veloce.

Il modo per uscire da questo dilemma è quello di usare il TLB; se il TLB può mantenere tutte le pagine utilizzate di frequente, la traduzione risulterà veloce come se si utilizzassero tabelle delle pagine regolari. Per quanto riguarda le miss di TLB, però, la tabella delle pagine invertite deve essere controllata dal software; un modo realizzabile per eseguire questa ricerca è quello di avere una tabella hash indicizzata sull'indirizzo virtuale: tutte le pagine virtuali correntemente in memoria che hanno lo stesso valore d'indice sono concatenate insieme, come mostrato in Figura 4.15. Se la tabella hash ha tanti slot quante pagine fisiche ha la macchina, la catena sarà mediamente lunga un solo elemento, velocizzando enormemente la traduzione. Una volta trovato il numero di pagina fisica, la nuova coppia (virtuale, fisica) verrà inserita nel TLB.

Le tabelle delle pagine invertite sono utilizzate specialmente su alcune stazioni di lavoro IBM ed Hewlett-Packard e diventeranno molto comuni, via via che le macchine a 64 bit si diffonderanno. Altri approcci per la gestione delle memorie virtuali di grandi dimensioni possono essere trovati in (Huck e Hays, 1993; Talluri e Hill, 1994; e Talluri ed altri, 1995).

4.4 Algoritmi di rimpiazzamento delle pagine

Quando si verifica un fault di pagina, il sistema operativo deve scegliere una pagina da rimuovere dalla memoria, per fare spazio alla pagina che deve essere caricata. Se la pagina che deve essere rimossa è stata modificata durante la sua permanenza in memoria, deve essere riscritta sul disco per rendere quella copia aggiornata; se, invece, la pagina non è stata modificata (per esempio, una pagina contenente il testo di un programma), la copia sul disco è già aggiornata e quindi non è necessaria alcuna riscrittura. La pagina che deve essere caricata viene semplicemente scritta sopra la pagina destinata ad essere scaricata.

Sebbene sia possibile, ad ogni fault di pagina, scegliere a caso una pagina da scaricare, le prestazioni del sistema sono molto migliori quando sceglio una pagina che non viene usata frequentemente, in caso contrario essa dovrà probabilmente essere ricaricata entro breve tempo, provocando ulteriore perdita di tempo. Sugli algoritmi per il rimpiazzamento delle pagine è stato fatto molto lavoro, sia teorico sia sperimentale; nelle sezioni che seguono descriveremo alcuni degli algoritmi più importanti.

Notiamo che il problema del "rimpiazzamento delle pagine" avviene anche in altri contesti di progettazione di un computer; per esempio, la maggior parte dei computer hanno una o più cache di memoria che contengono i blocchi di memoria, di 32 byte o di 64 byte, utilizzati di recente; quando la cache è piena, deve essere selezionato qualche blocco da rimuovere. Questo problema è esattamente lo stesso del rimpiazzamento delle pagine, eccetto che per una minore scala di tempo (deve essere fatto in pochi nanosecondi, non in millisecondi come nel caso del rimpiazzamento delle pagine). La ragione per la scala di tempo ridotta è che le miss dei blocchi sulla cache sono soddisfatte dalla memoria principale, che non ha tempi di ricerca né latenza rotazionale.

Un secondo esempio si può avere in un server Web: il server può mantenere nella sua cache di memoria un certo numero di pagine Web utilizzate di frequente, tuttavia, quando la cache di memoria è piena e viene riferita una nuova pagina, bisogna decidere qua-

le pagina Web scaricare. Le considerazioni sono simili a quelle fatte per le pagine della memoria virtuale, ad eccezione del fatto che le pagine Web non sono mai modificate nella cache, quindi hanno sempre una copia aggiornata sul disco; invece, in un sistema a memoria virtuale, le pagine nella memoria principale possono essere sporche o pulite (modificate o non modificate).

4.4.1 L'algoritmo ottimale di rimpiazzamento delle pagine

Il miglior algoritmo possibile per il rimpiazzamento delle pagine è facile da descrivere ma impossibile da implementare. Funziona così: nel momento in cui avviene un fault di pagina, in memoria si trova un certo insieme di pagine e a una di queste pagine si farà riferimento proprio nella prossima istruzione (la pagina contenente tale istruzione); altre pagine potrebbero non essere utilizzate per altre 10, 100 o forse 1000 istruzioni. Ogni pagina può essere etichettata con il numero di istruzioni che saranno eseguite prima che a quella pagina si faccia riferimento per la prima volta.

L'algoritmo ottimale di rimpiazzamento delle pagine dice semplicemente che dovrà essere rimossa la pagina con l'etichetta più alta. Se una pagina non sarà usata per 8 milioni di istruzioni e un'altra pagina non sarà usata per 6 milioni di istruzioni, rimuovere la prima posticipa il fault di pagina che la riporterà in memoria, il più lontano possibile nel futuro. I calcolatori, come le persone, cercano di posticipare eventi sgradevoli fin quando possibile.

L'unico problema di questo algoritmo è che è irrealizzabile: al momento del fault di pagina, il sistema operativo non ha nessun modo di sapere quando si farà riferimento a ciascuna delle pagine (abbiamo visto precedentemente una situazione simile, con l'algoritmo di schedulazione shortest job first: come fa il sistema a dire quale job è il più breve?). Nonostante ciò, facendo eseguire un programma su un simulatore e mantenendo traccia di tutti i riferimenti alle pagine, è possibile implementare il rimpiazzamento di pagina ottimale alla *seconda* esecuzione, usando le informazioni sui riferimenti alle pagine raccolte durante la *prima* esecuzione.

In questo modo è possibile confrontare le prestazioni degli algoritmi realizzabili con quella dell'algoritmo migliore possibile; se un sistema operativo raggiunge delle prestazioni, diciamo, peggiori soltanto dell'uno per cento rispetto all'algoritmo ottimale, gli sforzi spesi nel cercare un algoritmo migliore, produrranno al più un miglioramento dell'uno per cento.

Per evitare ogni possibile confusione, bisogna che sia chiaro che questa registrazione dei riferimenti delle pagine si riferisce solamente al programma misurato e quindi ad un solo specifico input; l'algoritmo di rimpiazzamento delle pagine da esso derivato è quindi specifico per quel programma e per quei dati in input. Sebbene questo metodo sia utile per valutare gli algoritmi di rimpiazzamento delle pagine, esso non è di nessuna utilità nei sistemi reali. Nel seguito, studieremo algoritmi che sono utili sui sistemi reali.

4.4.2 L'algoritmo di rimpiazzamento delle pagine “non usate di recente”

Per permettere al sistema operativo di raccogliere statistiche utili sulle pagine usate e su quelle non usate, la maggior parte dei calcolatori dotati di memoria virtuale ha due bit di stato associati a ciascuna pagina. R viene messo a 1 ogni volta che la pagina viene riferita (letta o scritta); M viene messo a 1 quando la pagina viene scritta (cioè modificata). Questi bit sono contenuti in ogni elemento delle tabelle delle pagine, come illustrato in

Figura 4.13. È importante comprendere che questi bit devono essere aggiornati ad ogni riferimento in memoria, quindi è essenziale che vengano assegnati dall'hardware. Una volta che un bit è stato messo a 1, rimane a 1 fino a che il sistema operativo lo rimette a 0 via software.

Se l'hardware non ha questi bit, essi possono esser simulati come segue: quando un processo viene avviato, tutti gli elementi della sua tabella delle pagine vengono marcati come non presenti in memoria. Non appena una qualunque pagina verrà richiesta, ci sarà un fault di pagina; il sistema operativo, allora, metterà ad 1 il bit R (nelle sue tabelle interne), cambierà l'elemento della tabella delle pagine per puntare alla pagina corretta, con modalità READ ONLY, e rieseguirà l'istruzione. Se la pagina viene successivamente scritta, si avrà un altro fault di pagina, che permetterà al sistema operativo di mettere a 1 il bit M e di cambiare la modalità della pagina a READ/WRITE.

I bit R e M possono essere usati come segue per costruire un semplice algoritmo di paginazione. Quando viene lanciato un processo, entrambi i bit (di tutte le sue pagine) sono messi a 0 dal sistema operativo; periodicamente (ad esempio ad ogni interruzione dal clock) il bit R viene rimesso a 0, per distinguere le pagine che non sono state utilizzate recentemente da quelle che lo sono state.

Quando avviene un fault di pagina, il sistema operativo ispeziona tutte le pagine e le divide in 4 categorie, in base ai valori correnti dei loro bit R e M:

Classe 0: non usata, non modificata

Classe 1: non usata, modificata

Classe 2: usata, non modificata

Classe 3: usata, modificata.

Sebbene le pagine in classe 1 sembrino, ad una prima occhiata, impossibili, se ne possono avere quando il bit R di una pagina di classe 3 viene messo a 0 in seguito ad una interruzione da clock; le interruzioni di clock non mettono a 0 il bit M perché questa informazione è necessaria per sapere se la pagina deve essere riscritta o meno sul disco. Porre a 0 il bit R ma non il bit M consente ad una pagina di diventare di classe 1.

L'algoritmo NRU (Not Recently Used, "non usata di recente") rimuove una pagina a caso della classe non vuota di numero più basso. In questo algoritmo è implicito che è meglio rimuovere una pagina modificata cui non si è fatto riferimento per almeno un periodo del clock (tipicamente 20 ms) invece che una pagina pulita che viene usata frequentemente. L'attrattiva principale dell'algoritmo NRU è che è facile da capire, abbastanza efficiente da implementare, e che dà delle prestazioni che, sebbene certamente non ottimali, risultano spesso soddisfacenti.

4.4.3 L'algoritmo di rimpiazzamento delle pagine FIFO

Un altro algoritmo di paginazione a basso overhead è l'algoritmo FIFO (First-In, First-Out, primo arrivato, primo ad uscire). Per illustrare come funziona, consideriamo un supermercato che abbia scaffali per mostrare esattamente k prodotti diversi. Un giorno, una qualche ditta introduce un nuovo cibo economico: uno yogurt organico, istantaneo, congelato e liofilizzato, che può essere rigenerato in un forno a microonde. È un successo immediato, così il nostro supermercato, che è di grandezza finita, si deve liberare di un prodotto vecchio per poter immagazzinare quello nuovo.

Una possibilità è quella di trovare il prodotto che il supermercato ha cominciato ad immagazzinare da più tempo (magari qualcosa che ha iniziato a vendere 120 anni fa) e liberarsene ritenendo che nessuno ne sia più interessato. In effetti il supermercato contie-

ne una lista concatenata di tutti i prodotti correntemente in vendita, nell'ordine in cui sono stati introdotti; il nuovo prodotto finisce in coda alla lista e quello in testa alla lista viene eliminato.

La stessa idea è applicabile ad un algoritmo di rimpiazzamento delle pagine; il sistema operativo mantiene una lista di tutte le pagine correntemente in memoria, dove la pagina in testa alla lista è la più vecchia e la pagina in coda è quella arrivata più di recente. Al momento del fault di pagina, la pagina in testa viene rimossa, e la nuova pagina viene aggiunta in coda alla lista. Quando viene applicato ai grandi magazzini, l'algoritmo FIFO potrebbe rimuovere la brillantina per baffi, ma potrebbe anche rimuovere farina, sale o burro, e lo stesso problema sorge quando lo si applica ai calcolatori. Per questo motivo, FIFO viene usato raramente nella sua forma pura.

4.4.4 L'algoritmo di rimpiazzamento delle pagine “della seconda opportunità”

Una semplice modifica all'algoritmo FIFO che evita il problema dovuto allo scaricamento di una pagina molto usata, è quello di controllare il bit *R* della pagina più vecchia. Se è a 0, la pagina è sia vecchia sia non usata, così può essere immediatamente rimpiazzata. Se il bit *R* vale 1, viene posto a 0, la pagina viene spostata alla fine della coda delle pagine, ed il suo tempo di caricamento viene aggiornato come se la pagina fosse appena arrivata in memoria. Poi si continua la ricerca.

Il modo di operare di questo algoritmo, detto della **seconda opportunità**, viene illustrato in Figura 4.16. In Figura 4.16(a) vediamo le pagine da *A* ad *H* mantenute in una lista concatenata, ordinata rispetto al tempo di arrivo in memoria.

Supponiamo che al tempo 20 si verifichi un fault di pagina. La pagina più vecchia è *A*, che è arrivata al tempo 0, quando il processo è stato mandato in esecuzione; se *A* ha il bit *R* a zero, essa viene scaricata dalla memoria o scrivendola sul disco (se è stata modificata) o semplicemente abbandonandola (se non è stata modificata). D'altra parte, se il bit *R* è ad 1, *A* viene messa in fondo alla lista ed il suo “tempo di caricamento” viene riscritto con il tempo corrente (20); inoltre, il bit *R* viene posto a 0 e la ricerca di una pagina adeguata continua dalla pagina *B*.

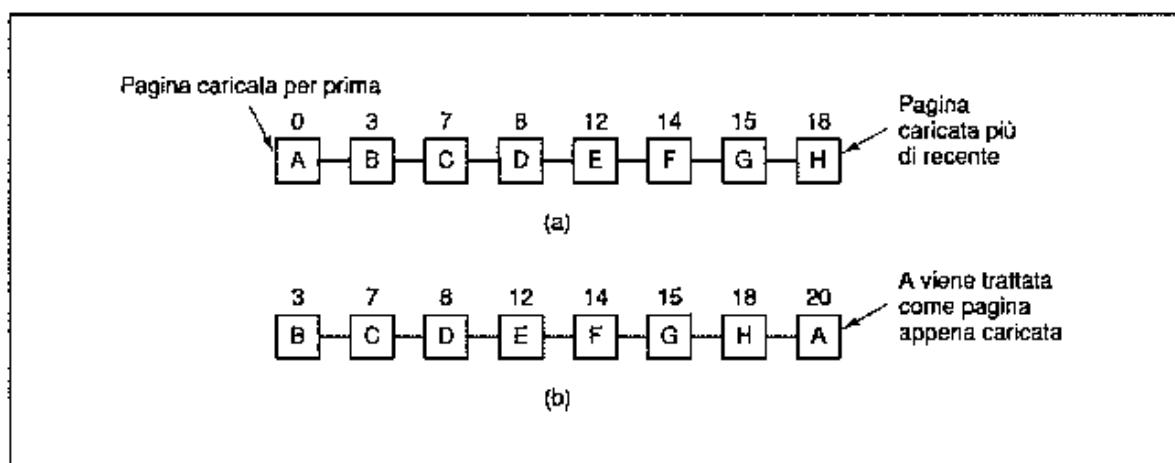


Figura 4.16 Funzionamento della seconda opportunità. (a) Pagine ordinate in modo FIFO. (b) La lista delle pagine quando al tempo 20 si verifica un fault e A ha il bit R messo a 1. I numeri sopra le pagine sono i relativi tempi di caricamento.

Ciò che la tecnica della seconda opportunità fa è di cercare una pagina vecchia che non sia stata usata nel precedente periodo del clock. Se tutte le pagine sono state usate, allora la tecnica della seconda opportunità degenera e diventa uguale alla FIFO; infatti, immaginate che tutte le pagine della Figura 4.16(a) abbiano il loro bit R messo a 1; il sistema operativo sposta le pagine una per una in fondo alla lista, mettendo a 0 il bit R ogni volta che una pagina viene spostata in fondo alla lista. Alla fine, ritorna alla pagina A , che questa volta ha il bit R messo a 0, e quindi a questo punto la pagina A viene scaricata. In questo modo l'algoritmo termina sempre.

4.4.5 L'algoritmo di rimpiazzamento delle pagine "dell'orologio"

Sebbene l'algoritmo della seconda opportunità sia un algoritmo ragionevole, esso risulta inefficiente dal momento che sposta continuamente pagine in fondo alla lista. Un approccio migliore è quello di mantenere le pagine in una lista circolare a forma di orologio, come mostrato in Figura 4.17; una lancetta punta alla pagina più vecchia.

Quando si verifica un fault di pagina, si controlla la pagina puntata dalla lancetta. Se il suo bit R è a 0, la pagina viene scaricata, la pagina nuova viene inserita al suo posto nella disposizione ad orologio e la lancetta viene spostata in avanti di una posizione; se il bit R è a 1, viene messo a zero e la lancetta viene spostata in avanti di una posizione. Questo processo viene ripetuto fino a che si trova una pagina con il bit R messo a 0. L'algoritmo viene detto, non sorprendentemente, dell'orologio, esso differisce dall'algoritmo della seconda opportunità solo per l'implementazione.

4.4.6 L'algoritmo di rimpiazzamento delle pagine LRU

Una buona approssimazione dell'algoritmo ottimo si basa sull'osservazione che le pagine che sono state frequentemente usate nelle ultime istruzioni lo saranno probabilmente anche nelle prossime istruzioni, mentre le pagine che non sono state usate per lunghissimo tempo non lo saranno ancora per molto tempo. L'idea suggerisce un algoritmo che si può effettivamente realizzare: quando si verifica un fault di pagina, buttare via la pagina che non è stata usata da più lungo tempo. Questa strategia è detta paginazione LRU (Least Recently Used, usata meno di recente).

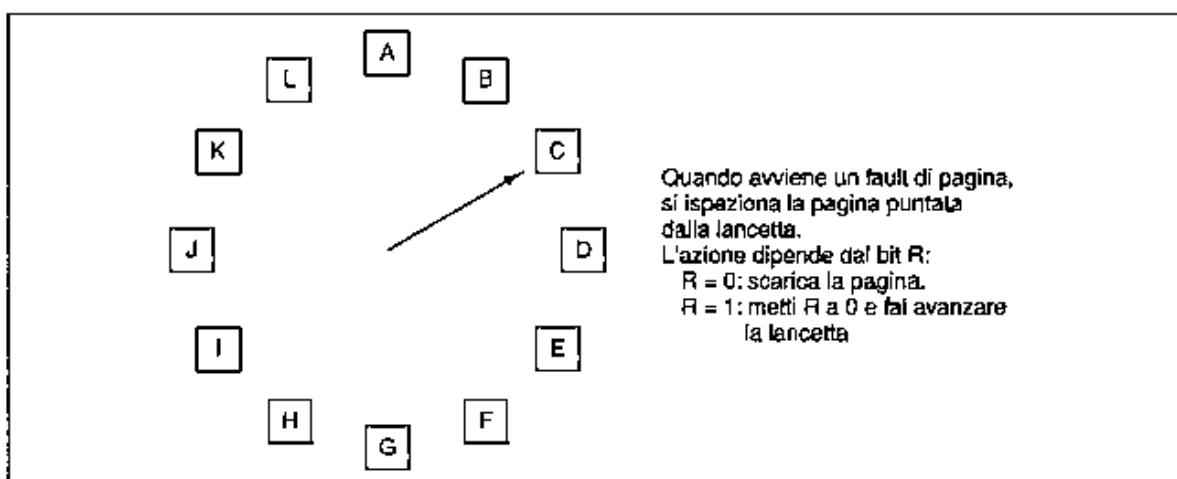


Figura 4.17 L'algoritmo di rimpiazzamento delle pagine "dell'orologio".

Sebbene LRU sia realizzabile dal punto di vista teorico, non è economica. Per implementare completamente LRU è necessario mantenere liste concatenate di tutte le pagine in memoria, con la pagina usata più recentemente in testa alla lista, e quella usata meno di recente in fondo; la difficoltà sta nel fatto che la lista deve essere aggiornata ad ogni riferimento alla memoria. Trovare una pagina nella lista, cancellarla e poi reinserirla in testa alla lista è un'operazione che richiede molto tempo; a meno che non si disponga di hardware speciale (sempre che tale hardware possa essere costruito).

Tuttavia, vi sono altri modi per implementare l'algoritmo LRU con hardware speciale. Consideriamo il modo più semplice. Questo metodo richiede che l'hardware sia dotato di un contatore a 64 bit, C, che viene incrementato automaticamente dopo ogni istruzione; inoltre ogni elemento della tabella delle pagine deve avere anche un campo abbastanza grande da contenere il contatore. Dopo ogni riferimento in memoria, il valore corrente di C viene memorizzato nell'elemento della tabella delle pagine corrispondente alla pagina appena usata. Quando si verifica un fault di pagina, il sistema operativo esamina tutti i contatori nella tabella delle pagine, per trovare quello con valore più basso: la pagina corrispondente è quella usata meno di recente.

Diamo ora un'occhiata ad un secondo algoritmo LRU realizzato in hardware. Per una macchina con n pagine fisiche, l'hardware LRU può mantenere una matrice di $n \times n$ bit, inizialmente tutti a 0; ogni volta che viene riferita la pagina fisica k , l'hardware mette prima tutti i bit della riga k a 1, poi tutti i bit della colonna k a 0. In ogni istante, la riga con il più piccolo valore binario è quella usata meno di recente, la riga con il valore immediatamente più alto è la successiva usata meno di recente, la riga con il valore immediatamente più alto è la successiva usata meno di recente, e così via. Il funzionamento di questo algoritmo è illustrato in Figura 4.18 per quattro pagine fisiche e con le pagine usate nell'ordine

0 1 2 3 2 1 0 3 2 3

Dopo che la pagina 0 è stata riferita, abbiamo la situazione di Figura 4.18(a). Dopo che la pagina 1 è stata riferita, abbiamo la situazione di Figura 4.18(b), e così via.

Pagina	Pagina	Pagina	Pagina	Pagina
0	0	0	0	0
0 0 1 1 1	0 0 1 1 1	0 0 0 1 1	0 0 0 0 0	0 0 0 0 0
1 0 0 0 0	1 0 1 1 1	1 0 0 1 1	1 0 0 0 0	1 0 0 0 0
2 0 0 0 0	0 0 0 0 0	1 1 0 1 1	1 1 0 0 0	1 1 0 1 1
3 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 1 1 0 0	1 1 0 0 0

(a)	(b)	(c)	(d)	(e)
0 0 0 0 0	0 1 1 1 1	0 1 1 0 0	0 1 0 0 0	0 1 0 0 0
1 0 1 1 1	0 0 1 1 1	0 0 1 0 0	0 0 0 0 0	0 0 0 0 0
1 0 0 1 1	0 0 0 1 1	0 0 0 0 0	1 1 0 1 1	1 1 0 0 0
1 0 0 0 0	0 0 0 0 0	1 1 1 0 0	1 1 0 0 0	1 1 1 0 0

(f)	(g)	(h)	(i)	(j)
0 0 0 0 0	0 1 1 1 1	0 1 1 0 0	0 1 0 0 0	0 1 0 0 0
1 0 1 1 1	0 0 1 1 1	0 0 1 0 0	0 0 0 0 0	0 0 0 0 0
1 0 0 1 1	0 0 0 1 1	0 0 0 0 0	1 1 0 1 1	1 1 0 0 0
1 0 0 0 0	0 0 0 0 0	1 1 1 0 0	1 1 0 0 0	1 1 1 0 0

Figura 4.18 LRU che usa una matrice quando le pagine sono riferite nell'ordine 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

4.4.7 La simulazione della LRU via software

Sebbene i precedenti algoritmi LRU siano, in principio, entrambi realizzabili, poche macchine (se non nessuna) hanno a disposizione questo hardware, quindi essi sono di poco aiuto al progettista di un sistema operativo per una macchina che non possiede questo hardware. Al loro posto è necessaria una soluzione che possa essere implementata via software: una possibilità è l'algoritmo detto **non usata frequentemente o NFU** (Not Frequently Used). L'algoritmo richiede un contatore associato ad ogni pagina, inizialmente a 0; ad ogni interruzione dal clock, il sistema operativo esamina tutte le pagine in memoria e per ogni pagina, il bit R, 0 o 1, viene sommato al contatore. In effetti, i contatori sono un tentativo di tenere traccia della frequenza con cui si fa riferimento ad ogni pagina; quando si verifica un fault di pagina, viene scelta per il rimpiazzamento la pagina con il contatore più basso.

Il problema principale con l'algoritmo NFU è che non si dimentica mai di niente. Ad esempio, in un compilatore a più passate le pagine che sono state usate pesantemente durante la prima passata, possono continuare ad avere un valore del contatore alto anche nelle passate successive. Infatti, se la prima passata ha un tempo di esecuzione più lungo delle altre, le pagine contenenti il codice per le passate successive avranno sempre un contatore più basso delle pagine della prima passata. Di conseguenza, il sistema operativo rimuoverà pagine utili invece di pagine che non vengono più usate.

Fortunatamente una piccola modifica all'algoritmo NFU lo rende capace di simulare piuttosto bene l'algoritmo LRU. La modifica avviene in due fasi. Primo, ognuno dei contatori viene shiftato a destra di un bit prima di sommarvi il bit R; secondo, il bit R viene sommato al bit più a sinistra invece che a quello più a destra.

La Figura 4.19 illustra come funziona l'algoritmo modificato, conosciuto come **algoritmo dell'invecchiamento**. Supponiamo che dopo il primo ciclo di clock i bit R delle

Bit R per le pagine da 0 a 5 al ciclo 0	Bit R per le pagine da 0 a 5 al ciclo 1	Bit R per le pagine da 0 a 5 al ciclo 2	Bit R per le pagine da 0 a 5 al ciclo 3	Bit R per le pagine da 0 a 5 al ciclo 4
1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Pagina				
0	10000000	11000000	11100000	11110000
1	00000000	10000000	11000000	01100000
2	10000000	01000000	00100000	00100000
3	00000000	00000000	10000000	00100000
4	10000000	11000000	01100000	10110000
5	10000000	01000000	10100000	00101000
(a)	(b)	(c)	(d)	(e)

Figura 4.19 L'algoritmo dell'invecchiamento simula l'algoritmo LRU via software. Vengono mostrate sei pagine per cinque cicli di clock (da (a) a (e)).

pagine da 0 a 5 abbiamo i valori 1, 0, 1, 0, 1 e 1, rispettivamente (la pagina 0 a 1, la pagina 1 a 0, la pagina 2 a 1, eccetera). In altre parole, tra il ciclo 0 ed il ciclo 1, le pagine 0, 2, 4 e 5 sono state usate, ed i loro bit R messi ad 1, mentre quelli delle altre sono rimasti a 0. Dopo che i sei contatori corrispondenti sono stati shiftati ed il bit R inserito a sinistra, tali contatori hanno i valori mostrati in Figura 4.17(a); le quattro colonne restanti mostrano i sei contatori dopo i successivi quattro cicli di clock.

Quando si verifica un fault di pagina, viene rimossa la pagina con il contatore più basso. Chiaro che una pagina che non è stata usata per, diciamo, quattro cicli di clock, avrà quattro zeri all'estrema sinistra nel contatore, e quindi avrà un valore più basso del contatore di una pagina che non è stata usata per tre cicli di clock.

Questo algoritmo differisce dall'LRU in due modi. Consideriamo le pagine 3 e 5 in Figura 4.19(e). Nessuna delle due è stata usata negli ultimi due cicli di clock; entrambe sono state usate nel ciclo precedente i due. Secondo l'algoritmo LRU, se una pagina dovesse essere rimpiazzata dovremmo scegliere una di queste due pagine. Il problema è che non sappiamo quale di queste due è stata usata per ultima nell'intervallo tra il ciclo 1 ed il ciclo 2. Registrando soltanto un bit per ogni intervallo di tempo, abbiamo perso la possibilità di distinguere all'interno di un intervallo di clock i riferimenti avvenuti presto da quelli accaduti successivamente. Tutto ciò che possiamo fare è rimuovere la pagina 3, poiché la pagina 5 è stata riferita anche 2 cicli prima mentre la pagina 3 non lo è stata.

La seconda differenza tra l'algoritmo LRU e l'algoritmo dell'invecchiamento è che in quest'ultimo i contatori hanno un numero finito di bit, 8 bit in questo esempio. Supponiamo che due pagine abbiano ciascuna il contatore a 0: tutto ciò che possiamo fare è sceglierne una a caso. In realtà, potrebbe anche succedere che una delle pagine sia stata usata per l'ultima volta 9 cicli fa, e che l'altra sia stata usata l'ultima volta 1000 cicli fa; non abbiamo nessun modo per accorgercene. In pratica, comunque, 8 bit sono generalmente sufficienti se il ciclo di clock è di circa 20 ms: se una pagina non è stata usata in 160 ms, probabilmente non è così importante.

4.4.8 Il modello Working Set

Nella forma di paginazione più pura, i processi vengono fatti partire senza che nessuna delle loro pagine sia presente in memoria. Non appena la CPU tenta di prelevare la prima istruzione, avviene un fault di pagina che fa sì che il sistema operativo carichi la pagina che contiene la prima istruzione. Rapidamente, si hanno altri fault per le pagine delle variabili globali e dello stack, e dopo un po', il processo dispone della maggior parte delle pagine di cui ha bisogno e comincia a girare con un numero relativamente basso di fault di pagina. Questa strategia viene detta di **paginazione a richiesta**, dal momento che le pagine vengono caricate a richiesta, piuttosto che anticipatamente.

Naturalmente, è abbastanza facile scrivere un programma di prova che legge sistematicamente tutte le pagine appartenenti ad uno spazio di indirizzamento molto grande, e che causa talmente tanti fault di pagina che non vi è abbastanza memoria per soddisfare tutte le richieste. Per fortuna, la maggior parte dei processi non lavora in questo modo: essi mostrano una **località dei riferimenti**, il che significa che in ognuna delle fasi dell'esecuzione, il processo fa riferimento solo a una frazione relativamente piccola delle sue pagine. Ciascun passo di un compilatore a passate multiple, per esempio, utilizza solo una piccola frazione delle pagine, in generale diversa fra le varie fasi.

L'insieme delle pagine che vengono utilizzate correntemente da un processo viene detto **working set** o **insieme di lavoro** (Denning, 1968a; Denning, 1980). Se tutto l'insieme di lavoro fosse in memoria, il processo girerebbe senza causare molti fault di pagina fino al momento in cui passa ad un'altra fase di esecuzione (per esempio, un altro passo

di compilazione); se la memoria disponibile è troppo piccola per poter contenere tutto l'insieme di lavoro, il processo causerà molti fault di pagina e girerà molto lentamente, dal momento che per eseguire un'istruzione ci vogliono pochi nanosecondi e per caricare una pagina dal disco ci vogliono 10 millisecondi. Con un tasso di uno o due istruzioni ogni 10 millisecondi, ci vorrà un'eternità per completare l'esecuzione. Un programma che genera un fault di pagina dopo l'esecuzione di poche istruzioni viene detto in una situazione di **thrashing** (Denning, 1968b).

In un sistema a multiprogrammazione, i processi vengono frequentemente spostati sul disco (cioè, tutte le loro pagine vengono scaricate dalla memoria) per far sì che altri processi godano del loro turno di accesso alla CPU. Si pone la domanda di cosa fare quando un processo deve essere ricaricato: tecnicamente, non occorre fare nulla. Il processo provocherà semplicemente dei fault di pagina fino a che il suo working set non sarà stato caricato in memoria. Il problema è che avere 20, 100 o anche 1000 fault di pagina ogni volta che un processo viene caricato in memoria è piuttosto pesante, e spreca anche una considerevole quantità di tempo di CPU, dal momento che al sistema operativo occorrono alcuni millisecondi di tempo di CPU per elaborare un fault di pagina.

Di conseguenza, molti sistemi operativi cercano di tener traccia degli insiemi di lavoro di ciascun processo, e si assicurano che l'insieme di lavoro sia stato caricato in memoria prima di permettere ad un processo di riprendere l'esecuzione. Questo approccio viene detto **modello dell'insieme di lavoro o working set** (Denning, 1970) ed è stato pensato per ridurre in maniera sensibile il tasso dei fault di pagina. Caricare le pagine prima di lasciar andare in esecuzione il processo viene detto anche **prepaging** (pre-paginazione). Si noti che l'insieme di lavoro cambia nel tempo.

È noto da tempo che la maggior parte dei programmi non riferiscono il loro spazio di indirizzamento in maniera uniforme, ma che i riferimenti tendono a concentrarsi in un piccolo numero di pagine. Un riferimento in memoria può ricercare un'istruzione, può ricercare dei dati o può memorizzare dei dati. Ad ogni istante di tempo t , esiste un insieme costituito da tutte le pagine utilizzate dai k riferimenti in memoria più recenti, questo insieme, $w(k, t)$, è l'insieme di lavoro. Poiché quando $k = 1$ i riferimenti più recenti devono aver utilizzato tutte le pagine usate anche dai riferimenti più recenti con $k > 1$, e possibilmente anche altre, $w(k, t)$ è una funzione su k monotona non decrescente; il limite di $w(k, t)$, all'aumentare di k , è finito, poiché un programma non può riferire più pagine di quante ne possa contenere il suo spazio di indirizzamento, e pochi programmi utiliz-

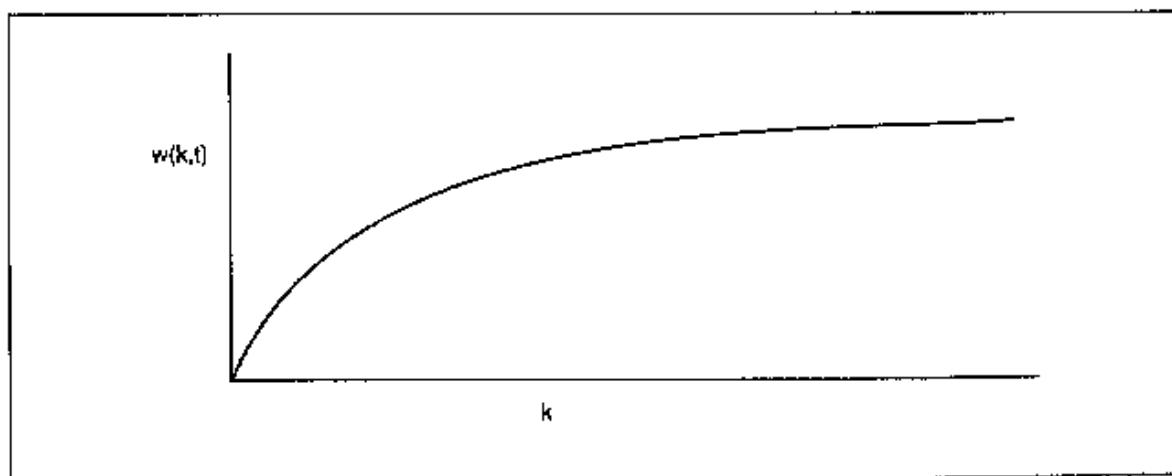


Figura 4.20 L'insieme di lavoro è l'insieme delle pagine utilizzate dai k riferimenti in memoria più recenti. La funzione $w(k, t)$ è la dimensione dell'insieme di lavoro al tempo t .

ranno ogni singola pagina. La Figura 4.20 mostra la dimensione dell'insieme di lavoro in funzione di k .

Il fatto che la maggior parte dei programmi accedono in maniera casuale ad un piccolo numero di pagine, ma che questo insieme cambia lentamente nel tempo, spiega la rapida crescita iniziale della curva, seguita da una lenta crescita per k grandi. Per esempio, un programma che sta eseguendo un ciclo su due pagine, che utilizza a sua volta dati su quattro pagine, può riferire tutte le sei pagine ogni 1000 istruzioni, ma il riferimento più recente a qualche altra pagina può essere avvenuto un milione di istruzioni prima, durante la fase di inizializzazione. A causa di questo comportamento asintotico, il contenuto dell'insieme di lavoro non è sensibile al valore di k scelto. In altre parole, esiste un grande intervallo di valori di k per i quali l'insieme di lavoro non è cambiato. Dato che l'insieme di lavoro varia lentamente nel tempo, si può supporre quali pagine saranno necessarie quando il programma verrà fatto ripartire, sulla base del contenuto del suo insieme di lavoro nel momento in cui era stato interrotto. La prepaginazione consiste nel caricare queste pagine prima che venga consentito al processo di girare nuovamente.

Per implementare il modello dell'insieme di lavoro, è necessario che il sistema operativo tenga traccia di quali pagine sono nell'insieme di lavoro; avere a disposizione questa informazione consente immediatamente di adottare un algoritmo per il rimpiazzamento delle pagine: quando si verifica un fault di pagina, si seleziona una pagina non presente nell'insieme di lavoro e la si scarica. Per implementare tale algoritmo abbiamo bisogno di un modo preciso per determinare quali pagine sono nell'insieme di lavoro e quali non lo sono, ad un certo istante di tempo.

Come dicevamo precedentemente, l'insieme di lavoro è l'insieme delle pagine utilizzate nei k riferimenti in memoria più recenti (altri autori usano i k riferimenti a pagina più recenti, ma la scelta è arbitraria). Per implementare l'algoritmo dell'insieme di lavoro, si deve scegliere in anticipo il valore di k ; una volta selezionato, dopo ogni riferimento in memoria, l'insieme delle pagine utilizzate dai precedenti k riferimenti in memoria è determinato in modo univoco.

Certamente, avere una definizione operativa dell'insieme di lavoro non significa avere un modo efficiente per monitorare tale insieme in tempo reale, durante l'esecuzione del programma. Si pensi ad esempio ad un registro a scorrimento di lunghezza k , dove ad ogni riferimento in memoria si sposta il registro di una posizione a sinistra, e viene inserito a destra il numero di pagina riferita più recentemente; l'insieme dei k numeri di pagina nel registro a scorrimento è l'insieme di lavoro. In teoria, ad ogni fault di pagina, i numeri contenuti nel registro possono essere letti e ordinati, si possono rimuovere le pagine duplicate, ed il risultato finale dovrebbe essere l'insieme di lavoro; tuttavia, mantenere il registro a scorrimento ed elaborarlo ad ogni fault di pagina è proibitivamente costoso, quindi questa tecnica non viene mai usata.

Vengono, invece, usate diverse approssimazioni. Una approssimazione utilizzata comunemente è quella di eliminare l'idea di contare all'indietro i k riferimenti in memoria e di usare invece il tempo di esecuzione. Per esempio, invece di definire l'insieme di lavoro come quelle pagine utilizzate durante i precedenti 10 milioni di riferimenti in memoria, possiamo definirlo come l'insieme delle pagine utilizzate durante i 100 msec precedenti di tempo di esecuzione; in pratica, tale definizione è ugualmente valida e molto più semplice da utilizzare. Si noti che per ogni processo, conta solo il suo tempo di esecuzione, perciò se un processo inizia a girare al tempo T e ha avuto 40 ms di tempo di CPU, al tempo reale $T + 100$ ms, per gli scopi dell'insieme di lavoro, il suo tempo è 40 ms. L'ammontare di tempo di CPU che un processo ha effettivamente utilizzato da quando è iniziato è spesso chiamato il suo **tempo virtuale corrente**; con questa approssimazione, l'insieme di lavoro di un processo è l'insieme delle pagine che ha riferito durante i precedenti τ secondi di tempo virtuale.

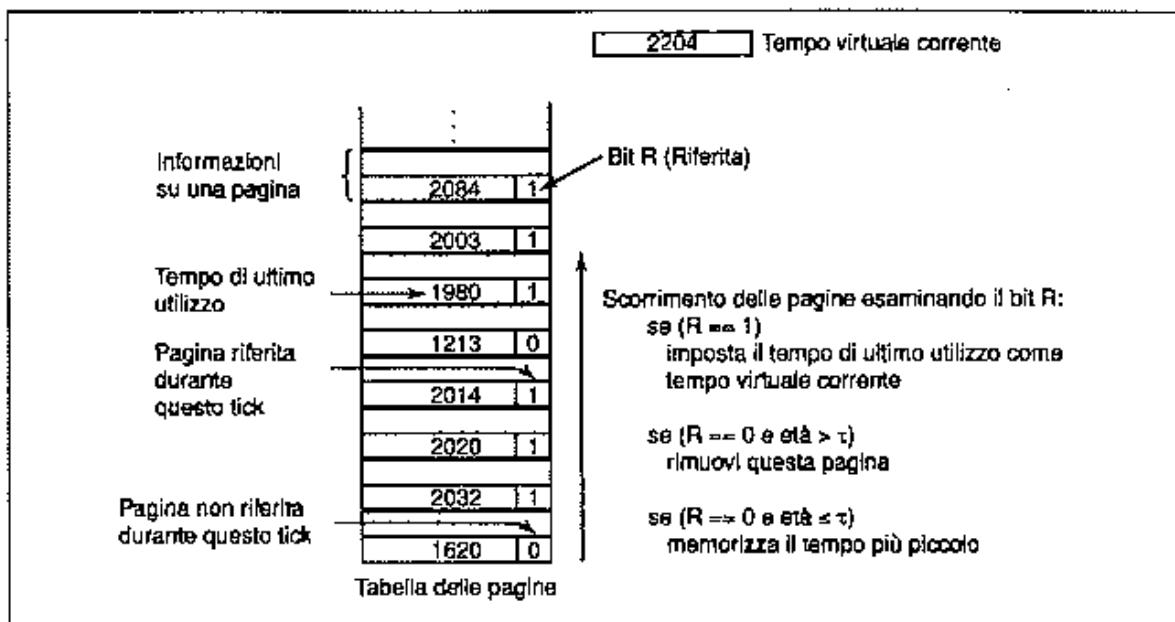


Figura 4.21 L'algoritmo dell'insieme di lavoro.

Ora osserviamo un algoritmo di rimpiazzamento delle pagine basato sul working set: l'idea di base è quella di trovare una pagina che non è nel working set e di scaricarla. In Figura 4.21 vediamo una porzione di una tabella delle pagine per una macchina; dato che vengono considerate come possibili candidate per l'eliminazione solo le pagine che sono in memoria, questo algoritmo ignora le pagine non presenti in memoria. Ogni elemento contiene (almeno) due dati per ogni pagina: il tempo approssimato a cui la pagina è stata usata per l'ultima volta, e il bit R (Riferita). Il rettangolo bianco vuoto rappresenta gli altri campi non necessari per questo algoritmo, come ad esempio il numero di pagina fisica, i bit di protezione, e il bit M (Modificata).

L'algoritmo funziona nel modo seguente: l'hardware si occupa di impostare i bit R e M, nel modo in cui abbiamo discusso precedentemente, in modo simile, viene effettuata periodicamente un'interruzione di clock in modo da consentire al software di porre a 0 il bit Riferita ad ogni tick del clock; ad ogni fault di pagina, la tabella delle pagine viene analizzata per cercare una pagina da scaricare.

Quando viene esaminato un elemento, viene esaminato anche il bit R: se assume il valore 1, il tempo virtuale corrente viene scritto nel campo Tempo di ultimo utilizzo presente nella tabella delle pagine, il quale indica che la pagina era in uso al momento in cui è avvenuto il fault di pagina, poiché la pagina è stata riferita durante il tick del clock corrente, è chiaramente nell'insieme di lavoro, e non è candidata ad essere rimossa (si suppone che τ misuri numerosi tick del clock).

Se R assume il valore 0, la pagina non è stata riferita durante il tick del clock corrente e può essere candidata alla rimozione; per vedere se deve essere rimossa o meno, viene calcolata la sua età, cioè, il suo tempo virtuale corrente meno il suo Tempo di ultimo utilizzo, e questa viene confrontata con τ , quindi se l'età è più alta di τ , la pagina non è più nell'insieme di lavoro, questa viene quindi eliminata e viene caricata la nuova pagina. La scansione continuerà aggiornando comunque gli elementi rimanenti.

Tuttavia, se R è 0 ma l'età è minore o uguale a τ , la pagina è ancora nell'insieme di lavoro, quindi viene temporaneamente risparmiata, ma la pagina con l'età maggiore (quella che ha il minor valore del Tempo di ultimo utilizzo) viene segnata. Se è stata analizzata tutta la tabella senza trovare un candidato da eliminare, questo significa che tutte le pagine sono

nell'insieme di lavoro, in questo caso, se vengono trovate una o più pagine con $R = 0$, viene eliminata quella con l'età maggiore. Nel caso peggiore, tutte le pagine sono state riferite durante il tick del clock corrente (e perciò tutte hanno $R = 1$), quindi ne viene scelta una a caso per la rimozione, preferibilmente, se presente, una pagina pulita (non modificata).

4.4.9 L'algoritmo di rimpiazzamento delle pagine WSClock

L'algoritmo base dell'insieme di lavoro è ingombrante poiché ad ogni fault di pagina deve essere analizzata tutta la tabella delle pagine, fino a quando non viene trovata una pagina candidata adatta. L'algoritmo WSClock (Carry e Heunessey, 1981) è un algoritmo perfezionato, basato sull'algoritmo dell'orologio ma che utilizza anche le informazioni dell'insieme di lavoro; per la sua semplicità di implementazione e per le buone prestazioni è utilizzato largamente nella pratica.

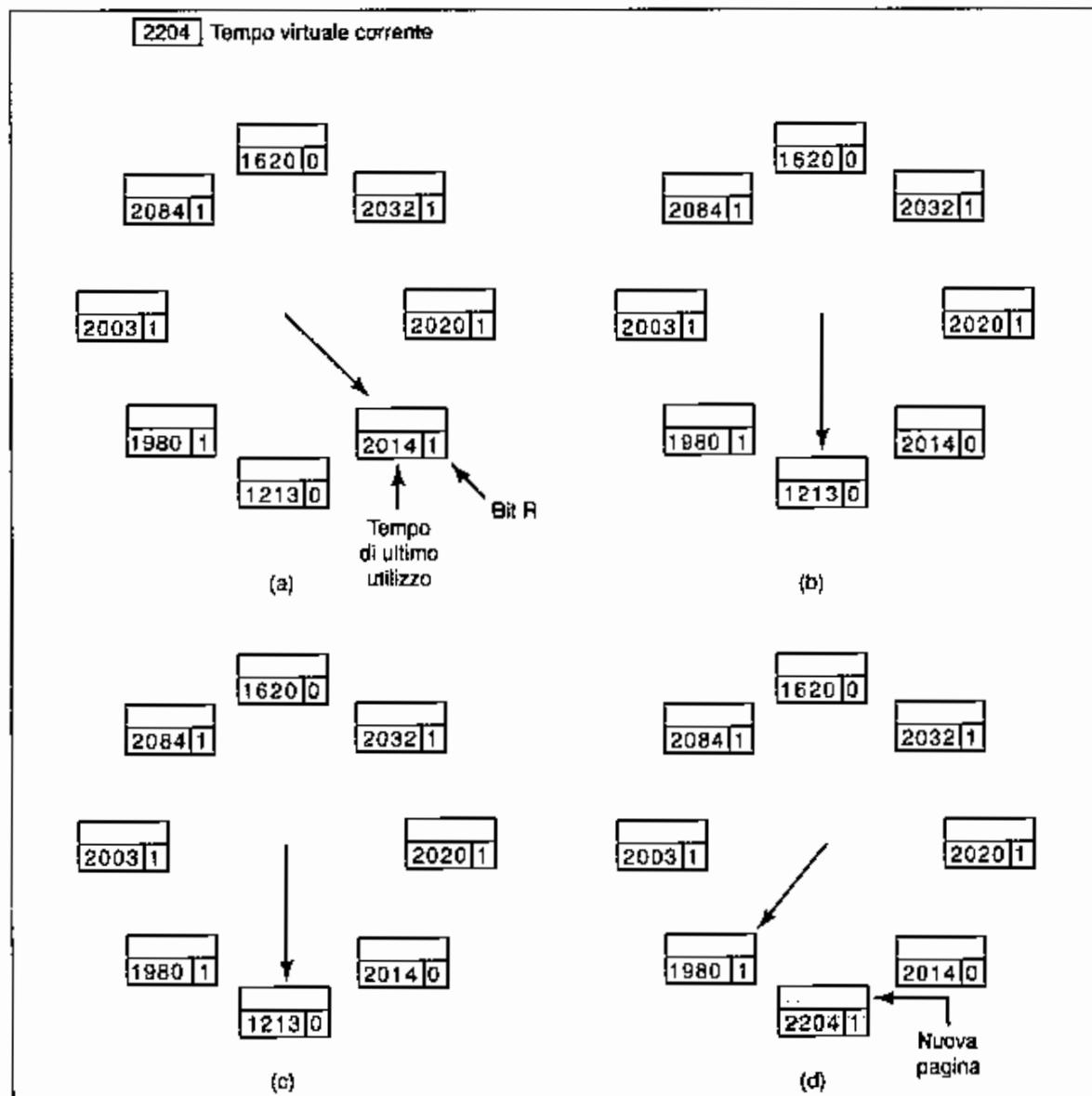


Figura 4.22 Funzionamento dell'algoritmo WSClock. (a) e (b) mostrano un esempio di cosa accade quando $R = 1$. (c) e (d) mostrano un esempio di cosa accade quando $R = 0$.

La struttura dati necessaria è una lista circolare di pagine fisiche, come nell'algoritmo dell'orologio, e come mostrato in Figura 4.22(a). Inizialmente, questa lista è vuota; quando viene caricata la prima pagina, essa viene aggiunta alla lista e ogni volta che viene caricata una pagina, questa va a finire nella lista in modo da formare un anello. Ogni elemento contiene il campo relativo al *Tempo di ultimo utilizzo* derivante dall'algoritmo base dell'insieme di lavoro, e contiene inoltre il bit *R* (mostrato in figura) ed il bit *M* (non mostrato in figura).

Come nell'algoritmo dell'orologio, ad ogni l'auft di pagina, la pagina puntata dalla lancetta viene esaminata per prima: se il bit *R* assume il valore 1, la pagina è stata usata durante il tick corrente, quindi non è una buona candidata ad essere rimossa, il bit *R* viene quindi impostato a 0, quindi si passa ad esaminare la pagina successiva e l'algoritmo viene ripetuto per quella pagina; in Figura 4.22(b) viene mostrato lo stato dopo questa sequenza di eventi.

Ora si consideri cosa accade se la pagina puntata dalla lancetta ha il bit *R* a 0, come mostrato in Figura 4.22(c). Se l'età è maggiore di τ e la pagina non è modificata, non è nell'insieme di lavoro ed esiste una sua copia valida su disco; la pagina fisica viene quindi semplicemente rilasciata e la nuova pagina viene inserita al suo posto, come mostrato in Figura 4.22(d). Viceversa, se la pagina è stata modificata (sporca), non può essere cancellata immediatamente poiché non abbiamo una copia valida sul disco. Per evitare un cambio di processo, la scrittura su disco viene schedulata, si avanza la lancetta e l'algoritmo procede con la pagina successiva. Dopo tutto, analizzando la pagina successiva l'algoritmo potrebbe trovare una vecchia pagina non modificata che potrebbe essere utilizzata immediatamente.

In linea di principio, tutte le pagine potrebbero essere schedurate per operazioni di I/O su disco in un giro di orologio; per ridurre il traffico su disco, si può impostare un limite, consentendo di riscrivere un massimo di n pagine; quando questo limite viene raggiunto, non viene schedulata nessuna nuova scrittura.

Cosa accade se la lancetta ritorna al punto di partenza? Vi sono due casi distinti:

1. È stata schedulata almeno una scrittura.
2. Non è stata schedulata nessuna scrittura.

Nel primo caso, la lancetta continua a girare in cerca di una pagina non modificata (pulita); visto che sono state schedolate una o più scritture, prima o poi qualche scrittura sarà completata, e la relativa pagina sarà marcata come non modificata. Viene quindi scaricata la prima pagina non modificata che si incontra; questa pagina non è necessariamente la prima scrittura schedulata, poiché il driver del disco può riordinare le scritture per ottimizzare le prestazioni del disco stesso.

Nel secondo caso, tutte le pagine sono nell'insieme di lavoro, altrimenti sarebbe stata schedulata almeno una scrittura. In mancanza di informazioni aggiuntive, la cosa più semplice da fare è di rilasciare una qualsiasi pagina non modificata (pulita) e di utilizzarla; la sua locazione potrebbe essere memorizzata durante il giro della lancetta. Se non esiste alcuna pagina non modificata, allora viene selezionata la pagina corrente e questa viene riscritta su disco.

4.4.10 Riepilogo degli algoritmi di rimpiazzamento delle pagine

A questo punto abbiamo analizzato una varietà di algoritmi per il rimpiazzamento delle pagine, in questa sezione li riepilogheremo brevemente. La lista degli algoritmi discussi è mostrata in Figura 4.23.

L'algoritmo ottimale rimpiazza la pagina che verrà riferita per ultima fra le pagine correnti, sfortunatamente, non c'è modo di determinare quale pagina sarà l'ultima, così in pratica questo algoritmo non può essere utilizzato, tuttavia può essere utilizzato come punto di riferimento per misurare gli altri algoritmi.

L'algoritmo NRU divide le pagine in quattro classi a seconda dello stato dei bit R e M . Viene selezionata in maniera casuale una pagina dalla classe con numero più basso. Questo algoritmo è facile da implementare, ma è molto rozzo, e ne esistono migliori.

FIFO tiene traccia dell'ordine in cui le pagine vengono caricate in memoria, mantenendo queste pagine in una lista concatenata. Rimuovere la pagina più vecchia diventa semplice, ma tale pagina potrebbe essere ancora in uso, quindi FIFO non è una buona scelta.

L'algoritmo della seconda opportunità è una modifica di FIFO che controlla se una pagina è in uso prima di rimuoverla, se fosse così, la pagina viene mantenuta nella lista. Questa modifica migliora notevolmente le prestazioni. L'algoritmo dell'orologio è una implementazione differente dell'algoritmo della seconda opportunità; ha le stesse prestazioni, ma impiega un po' meno tempo per eseguire l'algoritmo.

LRU è un algoritmo eccellente, ma non può essere implementato senza hardware speciale, quindi se non è disponibile questo tipo di hardware, non può essere utilizzato. NFU è un rozzo tentativo di approssimare l'algoritmo LRU. Non è un buon algoritmo, mentre l'algoritmo dell'invecchiamento è una migliore approssimazione di LRU e può essere implementato in maniera efficiente; tale algoritmo è una buona scelta.

Gli ultimi due algoritmi utilizzano l'insieme di lavoro. L'algoritmo dell'insieme di lavoro ha delle prestazioni ragionevoli, ma è talvolta dispendioso da implementare. L'algoritmo WSClock è una variante che non solo fornisce delle buone prestazioni ma è anche efficiente da implementare.

In conclusione, i due algoritmi migliori sono l'algoritmo dell'invecchiamento e l'algoritmo WSClock. Essi sono basati, rispettivamente, sugli algoritmi LRU e working set, ed entrambi danno buone prestazioni in termini di paginazione e possono essere implementati in maniera efficiente. Esistono anche altri algoritmi, ma questi due, nella pratica, sono probabilmente i più importanti.

Algoritmo	Criteri
Ottimale	Non implementabile, ma utilizzato come parametro di confronto
NRU (non usata di recente)	Molto rozzo
FIFO (primo arrivato, primo ad uscire)	Potrebbe eliminare pagine importanti
Della seconda opportunità	Molto migliore rispetto a FIFO
Dell'orologio	Realistico
LRU (usata meno di recente)	Eccellente, ma difficile da implementare esattamente
NFU (non usata frequentemente)	Approssimazione rozza di LRU
Dell'invecchiamento	Algoritmo efficiente che approssima bene LRU
Del working set (insieme di lavoro)	Talvolta dispendioso da implementare
WSClock	Algoritmo valido ed efficiente

Figura 4.23 Algoritmi di rimpiazzamento delle pagine discussi nel testo.

4.5 Modellazione degli algoritmi di paginazione

Nel corso degli anni si è lavorato per modellare gli algoritmi di rimpiazzamento delle pagine da un punto di vista teorico. In questa sezione discuteremo alcune di queste idee, solo per vedere come funziona il processo di modellazione.

4.5.1 L'anomalia di Belady

Intuitivamente potrebbe sembrare che più pagine fisiche ha una memoria, meno fault di pagina avrà un programma. In maniera abbastanza sorprendente, questo non è sempre vero: Belady ed altri (1969) scoprirono un controsenso in cui l'algoritmo FIFO provoca più fault di pagina con 4 pagine fisiche che con 3. Questa strana situazione è diventata nota come **anomalia di Belady**; essa è illustrata in Figura 4.24 per un programma con 5 pagine virtuali numerate da 0 a 4. Le pagine sono usate nell'ordine

0 1 2 3 0 4 0 1 2 3 4

In Figura 4.24(a) vediamo come con tre pagine fisiche viene generato un totale di nove fault di pagina. In Figura 4.24(b) otteniamo dieci fault di pagina con quattro pagine fisiche.

4.5.2 Gli algoritmi a pila

Molti ricercatori informatici rimasero sbigottiti dall'anomalia di Belady e cominciarono a studiarla. Questo lavoro condusse allo sviluppo di un'intera teoria sugli algoritmi di paginazione e sulle loro proprietà. Sebbene la maggior parte di questo lavoro vada al di

Tutte le pagine fisiche inizialmente vuote													
Pagina più giovane	0	1	2	3	0	1	4	0	1	2	3	4	
	0	1	2	3	0	1	4	4	4	2	3	3	
		0	1	2	3	0	1	1	1	4	2	2	
Pagina più vecchia		0	1	2	3	0	0	0	1	4	4		
P	P	P	P	P	P	P	P	P	P	P	P		
													9 Fault di pagina

(a)

Tutte le pagine fisiche inizialmente vuote													
Pagina più giovane	0	1	2	3	0	1	4	0	1	2	3	4	
	0	1	2	3	3	3	4	0	1	2	3	4	
		0	1	2	2	2	3	4	0	1	2	3	
Pagina più vecchia		0	1	1	1	2	3	4	0	1	2		
P	P	P	P	P	P	P	P	P	P	P	P	P	10 Fault di pagina

(b)

Figura 4.24 L'anomalia di Belady. (a) FIFO con tre pagine fisiche. (b) FIFO con quattro pagine fisiche. Le P mostrano quali riferimenti alle pagine provocano fault di pagina.

la degli scopi di questo libro, nel seguito ne daremo una breve introduzione; per maggiori dettagli, si veda Maekawa ed altri (1987).

Tutta questa attività comincia con l'osservazione che ogni processo genera una sequenza di riferimenti alla memoria durante la sua esecuzione, ed ogni riferimento in memoria corrisponde ad una specifica pagina virtuale. Quindi, concettualmente, gli accessi in memoria di un processo possono essere caratterizzati da una lista (ordinata) di numeri di pagina, che viene detta **stringa dei riferimenti** e svolge un ruolo centrale della teoria. Per semplicità, nel resto di questa sezione consideriamo soltanto il caso di una macchina con un solo processo, in modo che ogni macchina abbia un'unica stringa dei riferimenti deterministica (con più processi avremmo dovuto tener conto dell'interallacciamento delle loro stringhe dei riferimenti, dovuto alla multiprogrammazione).

Un sistema di paginazione può essere caratterizzato da tre elementi:

1. La **stringa dei riferimenti** del processo in esecuzione.
2. L'**algoritmo di rimpiazzamento delle pagine**.
3. Il **numero di pagine fisiche disponibili in memoria**, m .

Concettualmente, possiamo immaginare un interprete astratto che funzioni come segue. Esso mantiene un vettore interno, M , che tiene traccia dello stato della memoria. Il vettore ha tanti elementi quante sono le pagine virtuali del processo, che chiameremo n . Il vettore M è diviso in due parti: la parte superiore, con m elementi, contiene tutte le pagine correntemente in memoria, la parte inferiore, con $n - m$ pagine, contiene tutte le pagine che sono già state usate, ma che poi sono state rimosse e non sono correntemente in memoria. Inizialmente M è l'insieme vuoto, poiché nessuna pagina è in memoria, e nessuna è stata (ancora) usata.

All'inizio dell'esecuzione il processo comincia ad aggiungere le pagine nella stringa dei riferimenti, una alla volta, e non appena una pagina viene aggiunta, l'interprete controlla se è in memoria (cioè nella parte superiore di M). Se non lo è, si verifica un fault di pagina; se c'è un elemento libero in memoria (cioè la parte superiore di M contiene meno di m elementi), la pagina viene caricata ed inserita nella parte superiore di M . Questa situazione si verifica solamente all'inizio dell'esecuzione; se la memoria è piena (cioè la parte superiore di M contiene m elementi) viene invocato l'algoritmo di rimpiazzamento delle pagine per rimuovere una pagina dalla memoria. Nel modello, succede che un'pagina viene spostata dalla parte superiore di m alla parte inferiore, e la pagina richiesta entra nella parte superiore; inoltre, la parte superiore e quella inferiore possono essere riorganizzate separatamente.

Per rendere più chiare le operazioni dell'interprete, vediamo un esempio concreto usando il rimpiazzamento delle pagine LRU. Lo spazio degli indirizzi virtuali ha otto pagine e la memoria fisica ne ha quattro. In alto nella Figura 4.25 abbiamo una stringa dei riferimenti formata dalle 24 pagine:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

Sotto la stringa dei riferimenti, abbiamo 25 colonne di otto elementi ciascuna. La prima colonna, che è vuota, riflette lo stato di M prima che inizi l'esecuzione; ogni colonna successiva mostra M dopo che una pagina è stata aggiunta per via di un riferimento, ed elaborata dall'algoritmo di paginazione. Il bordo in grassetto individua la parte superiore di M , cioè i primi quattro elementi, che corrispondono alle pagine fisiche in memoria; le pagine all'interno del rettangolo in grassetto sono in memoria e le pagine al disotto sono state scaricate su disco.

La prima pagina nella stringa dei riferimenti è 0 e quindi viene inserita nella parte alta

Stringa dei riferimenti	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4	
	0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3		
	0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7			
	0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	4	5	5			
	0	2	2	1	1	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6		
	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		
Fault di pagina	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P		
Stringa delle distanze	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Figura 4.25 Lo stato del vettore di memoria M , dopo l'elaborazione di ogni elemento della stringa dei riferimenti. La stringa delle distanze verrà discussa nella sezione successiva.

della memoria, come mostrato nella seconda colonna; la seconda pagina è 2 e viene aggiunta in testa alla terza colonna. Questa azione fa spostare verso il basso la pagina 0; in questo esempio, una nuova pagina caricata viene sempre inserita in testa e tutto il resto scende giù come richiesto.

Ciascuna delle prime sette pagine nella stringa dei riferimenti provoca un fault di pagina; i primi quattro possono essere trattati senza dover scaricare alcuna pagina, ma a partire dal riferimento alla pagina 5, il caricamento della pagina richiede lo scaricamento di una pagina vecchia.

Il secondo riferimento alla pagina 3 non provoca un fault dal momento che la pagina 3 è già presente in memoria, nondimeno l'interprete la rimuove dalla posizione dove si trovava per metterla in testa, come illustrato. Il processo continua per un po', finché non viene richiesta la pagina 5; questa viene spostata dalla parte bassa di M alla parte alta (cioè viene caricata da disco in memoria). Ogni volta che viene richiesta una pagina che non è dentro il rettangolo in grassetto, si verifica un fault di pagina, come indicato dalle P sotto la matrice.

Riassumiamo ora brevemente alcune delle proprietà di questo modello. Primo, quando si fa riferimento ad una pagina, essa viene sempre portata nell'elemento più alto di M . Secondo, se la pagina è già in M , tutte le pagine sopra di lei si spostano di una posizione verso il basso. Una transizione da dentro a fuori del rettangolo corrisponde ad una pagina rimossa dalla memoria. Terzo, le pagine che erano sotto la pagina richiesta non vengono spostate; in questo modo, il contenuto di M rappresenta esattamente il contenuto dell'algoritmo LRU.

Sebbene questo esempio usi LRU, il modello funziona ugualmente bene con altri algoritmi. In particolare c'è una classe di algoritmi che risultano più interessanti: gli algoritmi che hanno la proprietà

$$M(m, r) \subseteq M(m+1, r)$$

dove m rappresenta le pagine fisiche e r è un indice nella stringa dei riferimenti. La formula dice che l'insieme delle pagine incluse nella parte superiore di M con una memoria di m pagine fisiche, dopo r riferimenti in memoria, è incluso anche in M per una memoria con $m+1$ pagine fisiche. In altre parole, se incrementiamo la dimensione della memoria di una pagina fisica e rieseguiamo il processo, ad ogni punto durante l'esecuzione, tut-

te le pagine che erano nella prima esecuzione sono presenti anche nella seconda esecuzione, insieme con una pagina in più.

Dall'esame della Figura 4.25 e da una piccola riflessione sul suo funzionamento, dovrebbe essere chiaro che l'algoritmo LRU possiede questa proprietà. Anche alcuni altri algoritmi (ad esempio il rimpiazzamento ottimale di una pagina) la possiedono, ma l'algoritmo FIFO non la possiede. Gli algoritmi che hanno questa proprietà sono chiamati **algoritmi a stack**. Questi algoritmi non soffrono dell'anomalia di Belady e sono quindi molto amati dai teorici della memoria virtuale.

4.5.3 La stringa delle distanze

Per gli algoritmi a stack, spesso è conveniente rappresentare la stringa dei riferimenti in un modo più astratto dei veri numeri di pagina. Un riferimento ad una pagina, sarà rappresentato da qui in avanti con la distanza fra la testa della stack ed il punto dove la pagina era allocata. Per esempio, il riferimento alla pagina 1 nell'ultima colonna di Figura 4.25 è un riferimento ad una pagina a distanza 3 dalla testa dello stack (poiché la pagina 3 era in terza posizione *prima* del riferimento). Le pagine che non sono state ancora usate, e quindi non sono ancora sullo stack (cioè non ancora in M), sono dette essere a distanza ∞ . La stringa delle distanze per la Figura 4.25 è riportata in fondo alla figura stessa.

Da notare che la stringa delle distanze dipende non solo dalla stringa dei riferimenti, ma anche dall'algoritmo di paginazione. Con la stessa stringa dei riferimenti, un diverso algoritmo di paginazione farebbe scelte diverse sulla pagina da rimuovere e, di conseguenza, si avrebbe una diversa sequenza di stack.

Le proprietà statistiche della stringa delle distanze hanno un impatto importante sulle prestazioni dell'algoritmo. In Figura 4.26(a) vediamo la funzione densità di probabilità per gli elementi in una stringa delle distanze (fittizia) d : la maggior parte degli elementi nella stringa sono tra 1 e k . Con una memoria di k pagine fisiche si avrebbero pochi fault di pagina.

Viceversa, in Figura 4.26(b) i riferimenti sono così dispersi che l'unico modo per evitare un gran numero di fault di pagina è quello di dare al programma tante pagine fisiche quante sono le sue pagine virtuali. Avere un programma come questo è solo questione di sfortuna.

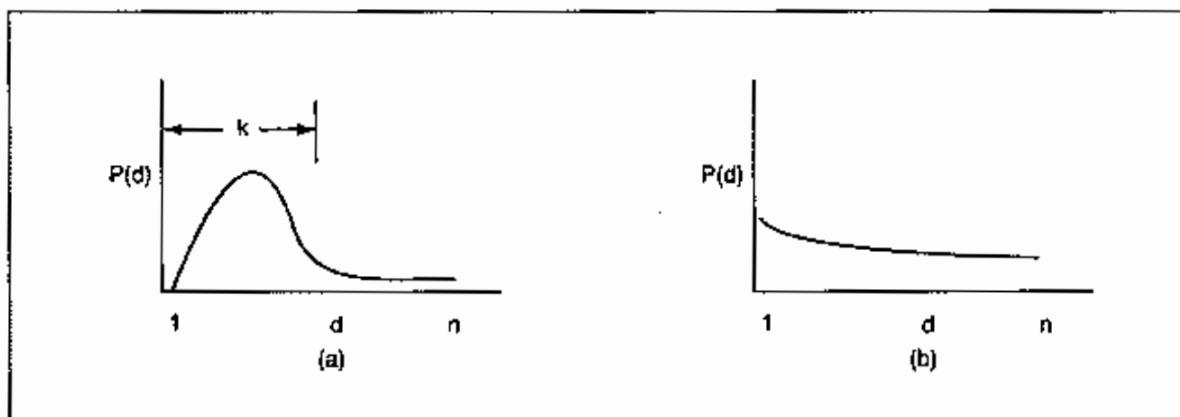


Figura 4.26. Funzione di densità di probabilità per due ipotetiche stringhe delle distanze.

4.5.4 La predizione dei tassi di fault di pagina

Una delle belle proprietà della stringa delle distanze è che può essere usata per predirre il numero di fault di pagina che si verificheranno con memorie di dimensioni diverse. Dimostreremo come si possa fare questo calcolo basandoci sull'esempio di Figura 4.25. L'obiettivo è di fare una passata di analisi della stringa delle distanze e, a partire dalle informazioni raccolte, di essere capaci di prevedere quanti fault di pagina avrebbe il processo in memoria con $1, 2, 3, \dots, n$ pagine fisiche, dove n è il numero di pagine virtuali nello spazio di indirizzamento del processo.

L'algoritmo parte esaminando la stringa delle distanze, pagina per pagina; esso tiene traccia delle occorrenze di 1, delle occorrenze di 2, e così via. Supponiamo che C_i sia il numero di occorrenze della distanza i ; per la stringa delle distanze della Figura 4.25, il vettore C è riportato in Figura 4.27(a). In questo esempio, succede quattro volte che la pagina richiesta sia già in testa allo stack; tre volte il riferimento è alla pagina successiva alla testa e così via. Supponiamo che C_∞ sia il numero di occorrenze di ∞ nella stringa delle distanze.

Calcoliamo ora il vettore F secondo la formula

$$F_m = \sum_{k=n+1}^m C_k + C_\infty$$

Il valore di F_m è il numero di fault di pagina che si verificheranno con la stringa delle distanze data ed m pagine fisiche. La Figura 4.27(b) riporta il vettore F per la stringa delle distanze di Figura 4.25. Ad esempio, $F_1 = 20$ e ciò indica che con una memoria contenente una sola pagina fisica, dei 24 riferimenti nella stringa tutti daranno fault di pagina tranne i quattro che sono gli stessi del riferimento di pagina precedente.

Per vedere perché questa formula funziona, ritorniamo al rettangolo in neretto in Figura 4.25. Supponiamo che m sia il numero di pagine fisiche nella parte superiore di M . Ogni volta che un elemento della stringa delle distanze è maggiore o uguale ad $m + 1$, si

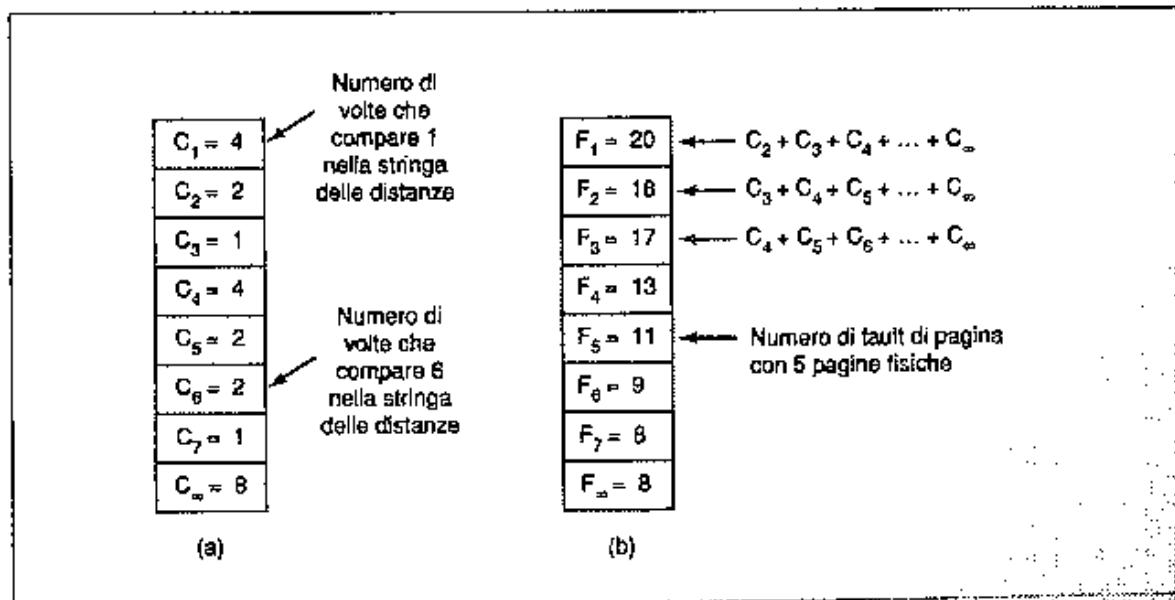


Figura 4.27 Calcolo del tasso di fault di pagina a partire dalla stringa delle distanze. (a) Il vettore C . (b) Il vettore F .

verifica un fault di pagina. La sommatoria nella formula precedente somma il numero di occorrenze di questi elementi. Questo modello può essere usato anche per fare altre predizioni (Maekawa ed altri, 1987).

4.6 Problematiche di progetto dei sistemi con paginazione

Nelle sezioni precedenti abbiamo spiegato come lavora la paginazione, abbiamo descritto alcuni fra gli algoritmi di rimpiazzamento delle pagine, ed abbiamo mostrato come modellarli. Tuttavia, conoscere semplicemente i meccanismi non basta: per progettare un sistema, dovete conoscere molto di più per poterlo fare funzionare bene; è la stessa differenza che c'è fra saper muovere la torre, il re, l'alfiere e gli altri pezzi degli scacchi ed essere un buon giocatore. Nelle sezioni che seguono, prenderemo in considerazione altri problemi che devono essere attentamente presi in considerazione dai progettisti dei sistemi operativi, per ottenere un sistema di paginazione con buone prestazioni.

4.6.1 Confronto delle politiche di allocazione globali e locali

Nelle sezioni precedenti abbiamo discusso alcuni algoritmi per scegliere le pagine da rimpiazzare quando si verifica un fault di pagina. Uno dei più grossi problemi associati a questa scelta (che per ora abbiamo accuratamente nascosto spazzandolo sotto il tapetto) sta nel decidere come vada allocata la memoria fra i processi in competizione per l'esecuzione.

Date un'occhiata alla Figura 4.28(a). In questa figura, l'insieme dei processi che possono essere in esecuzione è formato dai tre processi, A, B e C. Supponiamo che A generi

Età	(a)	(b)	(c)
A0			
10			
A1			
7			
A2			
5			
A3			
4			
A4			
6			
A5			
3	A6		
B0		B0	
9			
B1		B1	
4			
B2		B2	
6			
B3		B3	
2			
B4		B4	
5			
B5		B5	
6			
B6		B6	
12			
C1		C1	
3			
C2		C2	
5			
C3		C3	

Figura 4.28 Confronto fra rimpiazzamento delle pagine globale e locale. [a] Configurazione originale. [b] Rimpiazzamento delle pagine locali. [c] Rimpiazzamento delle pagine globali.

un fault di pagina. L'algoritmo per il rimpiazzamento delle pagine deve cercare la pagina usata meno di recente solo fra le sei pagine correntemente allocate al processo A, o deve invece ricercarla fra tutte le pagine in memoria? Se considera solo le pagine allocate ad A, la pagina di minore età è la A5, così si arriva alla situazione di Figura 4.28(b).

D'altra parte, se la pagina di minore età viene scaricata senza considerare a quale processo appartiene, verrebbe scelta la pagina B3 e si arriverebbe alla situazione della Figura 4.28(c). L'algoritmo della Figura 4.28(b) viene detto **algoritmo di rimpiazzamento locale**, mentre quello della Figura 4.28(c) viene detto **algoritmo globale**. Gli algoritmi locali sono quelli che assegnano ad ogni processo una quantità fissata di memoria; gli algoritmi globali allocano dinamicamente le pagine fisiche fra tutti i processi eseguibili; in questo modo, il numero delle pagine fisiche allocate ad un processo varia nel tempo.

In generale, gli algoritmi globali lavorano meglio, specialmente quando la dimensione dell'insieme di lavoro può variare durante la vita di un processo. Qualora venga usato un algoritmo locale e l'insieme di lavoro cresca, si potrà avere thrashing, anche nel caso in cui abbiamo ancora molte pagine fisiche libere; se l'insieme di lavoro si restringe, gli algoritmi locali sprecano memoria. Se si usano algoritmi globali, il sistema deve decidere in continuazione quante pagine fisiche assegnare ad ogni processo. Un modo è quello di controllare la dimensione dell'insieme di lavoro indicata dai bit dell'invecchiamento, ma questo metodo non elimina necessariamente il problema del thrashing; l'insieme di lavoro può cambiare dimensione nell'arco di pochi microsecondi, mentre i bit dell'invecchiamento rappresentano una misura grezza relativa ad un intervallo di vari cicli di clock.

Un altro approccio consiste nell'avere un algoritmo per l'allocazione delle pagine fisiche ai processi; un modo è quello di determinare periodicamente il numero dei processi in esecuzione e di allocare ad ogni processo la stessa quantità di pagine fisiche. Così, con 12416 pagine fisiche disponibili (cioè non allocate per il sistema operativo) e 10 processi, ogni processo avrebbe 1241 pagine e le rimanenti 6 pagine andrebbero in un pool dal quale si attinge quando si verifica un fault di pagina.

Sebbene questo metodo sembri equo, ha poco senso dare quote uguali di memoria ad un processo di 10KB e ad un processo di 300KB. Invece, le pagine possono essere allocate in misura proporzionale alla dimensione totale di ciascun processo, con il processo da 300KB che ottiene pagine per un totale che è pari a 30 volte le pagine allocate per il processo da 10KB. Probabilmente è saggio dare ad ogni processo una quota minima di pagine in modo che possa girare, indipendentemente dalla sua dimensione. Su alcune macchine, per esempio, una singola istruzione a due operandi può aver bisogno anche di sei pagine per essere eseguita, dal momento che l'istruzione, l'operando sorgente, e quello destinazione possono tutti e tre estendersi oltre il confine di una pagina (e quindi trovarsi parte in una pagina e parte nella pagina virtuale successiva); con un'allocazione di sole 5 pagine, programmi che contengono simili istruzioni non riuscirebbero a girare.

Se viene usato un algoritmo globale, si potrebbe cominciare ogni processo con un numero di pagina proporzionale alla dimensione del processo, ma l'allocazione deve essere aggiornata dinamicamente mentre il processo gira. Un modo per gestire l'allocazione è di utilizzare l'algoritmo PFF (Page Fault Frequency, frequenza dei fault di pagina). Esso indica quando incrementare o decrementare l'allocazione di una pagina del processo, ma non dice niente riguardo a quale pagina rimpiazzare quando si verifica un fault di pagina: controlla soltanto la dimensione allocata.

Per una classe molto vasta di algoritmi, tra cui l'algoritmo LRU, si sa che il tasso di fault decresce man mano che vengono assegnate più pagine, come discusso prima. Questa proprietà è illustrata in Figura 4.29.

Misurare il tasso dei fault di pagina è immediato: basta contare il numero dei fault per secondo, possibilmente calcolando anche la media degli ultimi secondi. Un modo sem-

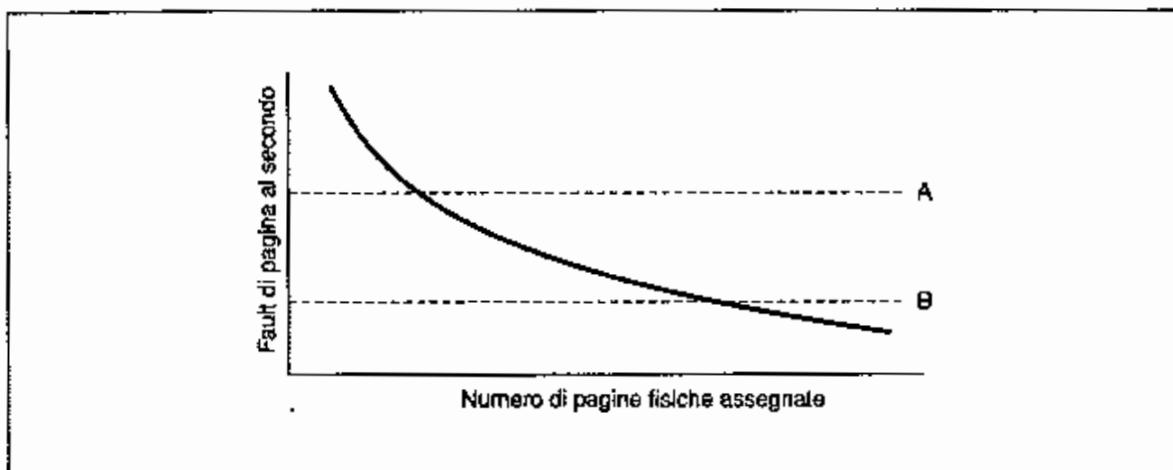


Figura 4.29 Tasso di fault di pagina come funzione delle pagine fisiche assegnate.

plice di eseguire questo è aggiungere il valore dell'ultimo secondo alla media corrente, e dividere il risultato per due. La linea tratteggiata indicata con *A* corrisponde ad un tasso di fault di pagina inaccettabilmente alto, quindi al processo vengono assegnate ulteriori pagine fisiche in modo da ridurre il tasso dei fault. La linea tratteggiata indicata con *B* corrisponde ad un tasso di fault di pagina così basso che si può concludere che a quel processo è stata assegnata troppa memoria; in questo caso, al processo possono essere tolte alcune pagine fisiche. In questo modo l'algoritmo PFF cerca di mantenere il tasso di fault di pagina entro limiti accettabili.

È importante sottolineare che alcuni algoritmi di rimpiazzamento delle pagine possono funzionare sia con una politica di rimpiazzamento locale sia con una politica di rimpiazzamento globale. Per esempio, FIFO può rimpiazzare la pagina più vecchia di tutta la memoria (algoritmo globale) oppure la pagina più vecchia posseduta dal processo corrente (algoritmo locale). In modo simile, LRU o alcune sue approssimazioni, possono rimpiazzare la pagina utilizzata meno di recente in tutta la memoria (algoritmo globale) oppure la pagina utilizzata meno di recente posseduta dal processo corrente (algoritmo locale). In alcuni casi la scelta fra locale e globale è indipendente dall'algoritmo.

D'altra parte, per altri algoritmi di rimpiazzamento delle pagine, può avere senso solo una strategia locale, in particolare, l'algoritmo dell'insieme di lavoro e l'algoritmo del WSClock si riferiscono ad un processo specifico e devono essere applicati in quel contesto. In realtà, non esiste un insieme di lavoro per la macchina nel suo complesso, e quindi cercare di utilizzare l'unione di tutti gli insiemi di lavoro farebbe perdere la proprietà di località e l'algoritmo non funzionerebbe più bene.

4.6.2 Controllo del carico

Anche utilizzando il migliore algoritmo di rimpiazzamento delle pagine e una allocazione globale ottimale delle pagine fisiche, può succedere che il sistema vada in thrash. Ciò capita ogni volta che l'insieme dei working set di tutti i processi supera la capacità di memoria. Un sintomo di questa situazione è dato dall'algoritmo PFF, quando indica che alcuni processi necessitano di più memoria ma che nessun processo necessita di meno memoria; in questo caso non c'è modo di fornire più memoria a questi processi che ne hanno bisogno, senza danneggiare qualche altro processo, e l'unica soluzione reale è quella di sbarazzarsi temporaneamente di alcuni processi.

Un modo per ridurre il numero dei processi in competizione per la memoria, è quello di spostare alcuni di loro su disco e di liberare tutte le pagine che essi detengono. Per esempio, un processo può essere spostato su disco e le sue pagine fisiche possono essere divise tra altri processi che stanno andando in thrashing; quindi se il thrashing termina, il sistema può continuare a girare per un po', se invece non termina, un altro processo deve essere spostato su disco, e così via, fino a quando non termina il thrashing. Perciò anche con la paginazione, è necessario uno swapper dei processi, solo che adesso lo swapping viene usato per ridurre la domanda potenziale di memoria, invece che per richiedere blocchi di memoria da utilizzare immediatamente.

Spostare i processi su disco per ridurre il carico in memoria ricorda la schedulazione a due livelli, in cui vengono spostati su disco alcuni processi e in cui viene usato uno scheduler a breve termine per schedulare i processi rimanenti. Chiaramente, le due idee possono essere combinate per scaricare su disco abbastanza processi e rendere il tasso dei fault di pagina accettabile. Periodicamente, alcuni processi vengono ricaricati dal disco e altri vengono scaricati al loro posto.

Tuttavia, un altro fattore da considerare è il grado di multiprogrammazione. Come abbiamo visto in Figura 4.4, quando il numero dei processi nella memoria principale è troppo basso, la CPU può essere inattiva per un considerevole periodo di tempo. Questa considerazione serve per raccomandare che, quando si decide quale processo togliere, non basta considerare soltanto la dimensione del processo e il tasso di paginazione, ma anche le sue caratteristiche, come ad esempio se sta usando pesantemente la CPU oppure l'I/O, ed anche quali caratteristiche hanno i processi rimanenti.

4.6.3 Dimensione della pagina

Spesso, la dimensione della pagina può essere scelta dal sistema operativo. Se anche l'hardware fosse stato progettato, per esempio, con pagine da 512 byte, il sistema operativo potrebbe facilmente trattare le pagine 0 e 1, 2 e 3, 4 e 5, eccetera come pagine da 1KB, allocando sempre due pagine fisiche da 512 byte consecutive.

Determinare la dimensione di pagina ottima richiede il bilanciamento di alcuni fattori contrapposti; come risultato, non esiste un ottimo generale. Per cominciare, vi sono due fattori che spingono per una dimensione piccola della pagina: un segmento scelto a caso, di testo di programma, di dati o della pila, non occupa un numero di pagine intero; in media, metà dell'ultima pagina sarà vuota, e lo spazio in più nella pagina viene sprecato. Questo spreco di spazio viene chiamato frammentazione interna. Con n segmenti in memoria e pagine di dimensione di p byte, $np/2$ byte risulteranno sprecati per via della frammentazione interna. Questo ragionamento depone a favore di una piccola dimensione delle pagine.

Un altro argomento a favore di pagine piccole balza agli occhi se pensiamo ad un programma composto da otto fasi sequenziali, ciascuna di 4KB. Con una dimensione delle pagine di 32KB, il programma deve tenere allocati 32KB per tutto il tempo; con una pagina da 16KB, necessita di soli 16KB, e con pagine da 4KB o più piccole, richiede ad ogni istante solo 4KB. In generale, una pagina di dimensioni grandi farà tenere in memoria una parte di programma non usato, rispetto ad una pagina di dimensioni più piccole.

D'altra parte, pagine di dimensione più piccola implicano la necessità di un maggior numero di pagine per il programma, e quindi tabelle delle pagine più grosse: un programma di 32KB richiede solo quattro pagine da 8KB, ma 64 pagine da 512 byte. I trasferimenti da e per il disco avvengono di solito una pagina alla volta, e la maggior parte del tempo viene speso per il movimento delle testine ed aspettando la rotazione del disco e, di conseguenza, il trasferimento di una pagina di piccole dimensioni richiede circa lo

stesso tempo del trasferimento di una pagina di grosse dimensioni. Ci potrebbero volere $64 * 10$ ms per caricare 64 pagine da 512 byte, ma solo $4 * 12$ ms per caricare quattro pagine da 8KB.

Su alcune macchine, la tabella delle pagine deve essere caricata in registri hardware ogni volta che la CPU passa da un processo ad un altro; su queste macchine, avere pagine di piccola dimensione significa che il tempo richiesto per caricare i registri di pagina aumenta man mano che la dimensione delle pagine diminuisce. Inoltre lo spazio occupato dalla tabella delle pagine cresce al decrescere della dimensione delle pagine.

Quest'ultimo argomento si può discutere da un punto di vista matematico. Sia s byte la dimensione media di un processo e p byte la dimensione delle pagine; inoltre, supponiamo che ciascun elemento della tabella delle pagine richieda e byte. Il numero di pagine approssimativamente richiesto per ognuno dei processi è dunque s/p ed esse occupano se/p byte nella tabella delle pagine. La memoria sprecata nell'ultima pagina per via della frammentazione interna è di $p/2$ byte. Così l'overhead totale dovuto alla tabella delle pagine e alla perdita dovuta alla frammentazione interna è data dalla

$$\text{overhead} = se/p + p/2$$

Il primo termine (dimensione della tabella delle pagine) è grande quando la dimensione delle pagine è piccola; il secondo termine (frammentazione interna) è grande quando la dimensione delle pagine è grande, quindi l'ottimo deve stare in qualche punto intermedio. Prendendo la derivata prima rispetto a p e ponendola a zero, otteniamo l'equazione

$$-se/p^2 + 1/2 = 0$$

Da questa equazione possiamo derivare una formula che dà la dimensione ottima della pagina (considerando soltanto la memoria sprecata in frammentazione e la dimensione della tabella delle pagine). Il risultato è:

$$p = \sqrt{2se}$$

Per $s = 1\text{MB}$ e $e = 8$ byte per ogni elemento della tabella delle pagine, la dimensione ottima della pagina è 4KB. La maggior parte dei calcolatori in commercio usano dimensioni delle pagine che variano da 512 byte a 64 KB. Un valore tipico utilizzato è 1KB, ma oggi sono più comuni 4KB o 8KB. Poiché le memorie diventano grandi, la dimensione della pagina tende a diventare anch'essa grande (ma non linearmente). Quadruplicare la dimensione della RAM raramente duplica la dimensione della pagina.

4.6.4 Spazi separati per istruzioni e dati

Molti calcolatori hanno uno spazio di indirizzamento singolo che contiene sia i programmi sia i dati, come mostrato in Figura 4.30(a), se questo spazio di indirizzamento è abbastanza grande, tutto funziona bene, tuttavia, spesso è troppo piccolo, e questo fatto costringe i programmatore a fare acrobazie per inserire ogni cosa nello spazio di indirizzamento.

Una soluzione, sviluppata per il PDP-11 (a 16 bit), è quella di avere spazi di indirizzamento separati per le istruzioni (testo del programma) e per i dati, chiamati rispettivamente **spazio-I** e **spazio-D**. Ogni spazio di indirizzamento va da 0 a qualche massimo, tipicamente $2^{16} - 1$ oppure $2^{32} - 1$; in Figura 4.30(b) li vediamo entrambi. Il linker deve conoscere quando devono essere utilizzati spazi I e D separati in quanto quando vengo-

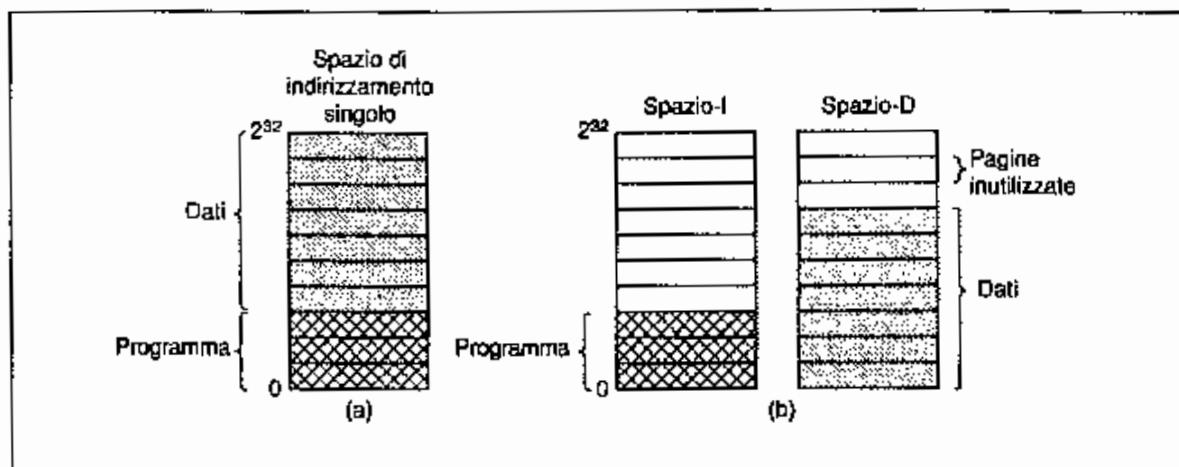


Figura 4.30 (a) Unico spazio di indirizzamento. (b) Spazio-I e spazio-D separati.

no utilizzati, i dati sono rilocati all'indirizzo virtuale 0 invece di iniziare dopo il programma.

In un calcolatore che utilizza questo modello, entrambi gli spazi di indirizzamento possono essere paginati in maniera indipendente; ognuno ha la sua tabella delle pagine, con la sua mappa delle pagine virtuali in pagine fisiche. Quando l'hardware vuole eseguire un'istruzione, esso sa che deve usare lo spazio-I e la tabella delle pagine dello spazio-I; in modo simile, i riferimenti ai dati devono essere inseriti nella tabella delle pagine dello spazio-D. Oltre a questa distinzione, avere spazi I e D separati non introduce nessuna complicazione speciale e duplica lo spazio di indirizzamento disponibile.

4.6.5 Pagine condivise

Un altro problema relativo all'implementazione è quello della condivisione. In un grosso sistema a multiprogrammazione, è situazione abbastanza comune quella in cui più utenti fanno girare lo stesso programma nello stesso momento. Per evitare di avere più copie della stessa pagina in memoria allo stesso istante risulta chiaramente più efficiente condividere le pagine. C'è un problema, e cioè che non tutte le pagine possono essere condivise; in particolare, le pagine a sola lettura, come le pagine del testo di un programma, possono essere condivise, mentre non possono essere condivise le pagine di dati.

Se vengono utilizzati spazi I e D separati, è abbastanza semplice condividere programmi avendo due o più processi che utilizzano la stessa tabella delle pagine per il proprio spazio-I, e tabelle delle pagine differenti per il proprio spazio-D. Tipicamente in una implementazione che supporta la condivisione in questo modo, le tabelle delle pagine sono strutture dati indipendenti dalla tabella dei processi; ogni processo quindi ha due puntatori nella sua tabella dei processi: uno per la tabella delle pagine dello spazio-I e uno per la tabella delle pagine dello spazio-D, come mostrato in Figura 4.31. Quando lo scheduler sceglie un processo da eseguire, utilizza questi puntatori per localizzare le tabelle delle pagine appropriate e per impostare la MMU. I processi possono condividere i programmi (o a volte le librerie) anche senza avere gli spazi I e D separati, ma il meccanismo è più complicato.

Quando due o più processi condividono del codice, si verifica un problema con le pagine condivise. Supponiamo che il processo A ed il processo B stiano entrambi facen-

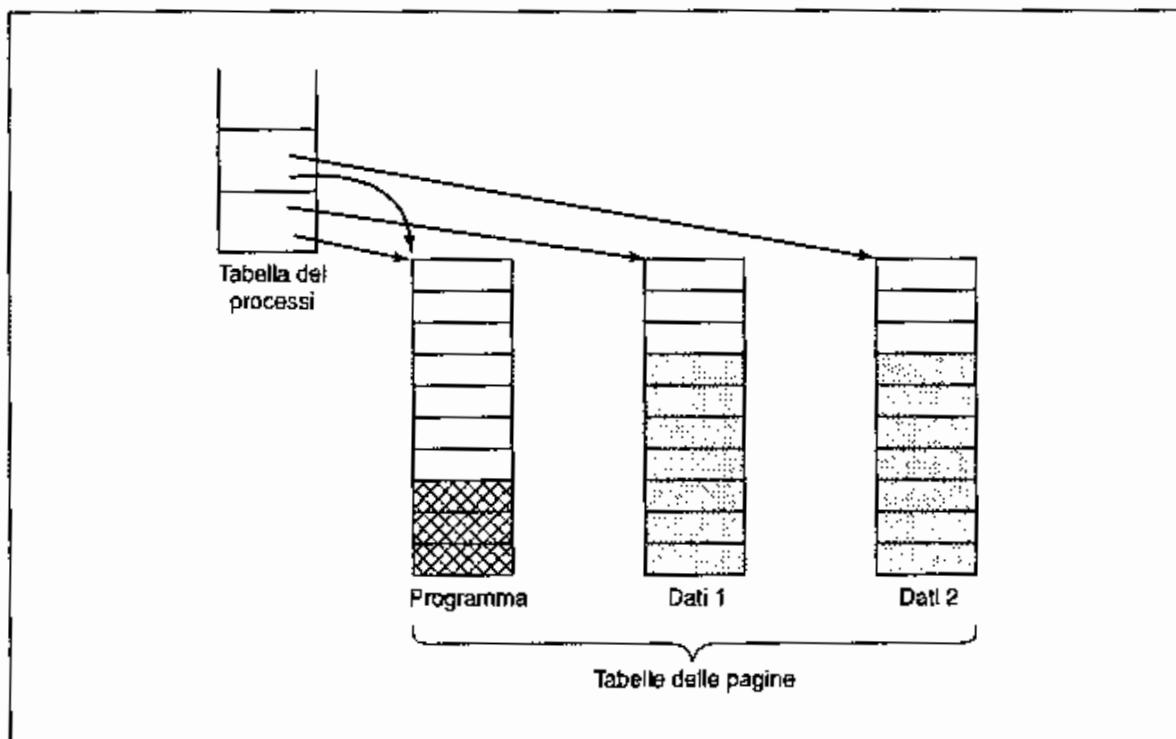


Figura 4.31 Due processi condividono lo stesso programma condividendo la sua tabella delle pagine.

do girare l'editor e condividano le sue pagine. Se lo scheduler decide di rimuovere il processo *A* dalla memoria, scaricando tutte le sue pagine e riempendo le pagine fisiche libere con qualche altro programma, il processo *B* genererà un alto numero di fault di pagina per ricaricare nuovamente quelle pagine.

In maniera del tutto analogica, quando il processo *A* termina è essenziale poter scoprire che le pagine sono ancora usate in modo che il loro spazio su disco non venga accidentalmente dichiarato libero. Dal momento che la ricerca su tutte le tabelle delle pagine per scoprire se una pagina è ancora in uso risulta di solito troppo costosa, sono necessarie strutture dati particolari che tengano traccia della condivisione delle pagine, specialmente se l'unità di condivisione è la pagina singola (o un insieme di pagine), invece che l'intera tabella delle pagine.

La condivisione dei dati è più complicata della condivisione del codice, ma non è impossibile. In particolare, in UNIX, dopo una chiamata di sistema fork, il processo padre e il processo figlio devono condividere il programma (testo e dati). In un sistema paginato, quello che spesso si fa è fornire a ciascuno di questi processi la propria tabella delle pagine, ed entrambe che puntano allo stesso insieme di pagine, perciò non viene fatta nessuna copiatura delle pagine nel momento in cui si esegue la fork, tuttavia, tutte le pagine dei dati sono mappate in entrambi i processi come READ ONLY.

Fin tanto che i processi leggono i loro dati, senza modificarli, questa situazione può continuare, ma non appena uno dei processi aggiorna una parola di memoria, la violazione della protezione read-only causa una trap al sistema operativo; viene quindi fatta una copia della pagina, in modo che ogni processo ora abbia la sua copia privata. Entrambe le copie vengono quindi impostate a READ-WRITE, in modo che successive scritture possano procedere senza ostacoli in entrambe le copie. Questa strategia implica che le pagine che non sono mai scritte (tra cui tutte le pagine del programma) non necessitano

di essere copiate, e solo le pagine dei dati che sono veramente scritte necessitano di essere copiate. Questo approccio, chiamato **copy-on-write**, cioè **copia quando scrivi**, migliora le prestazioni riducendo le copiature.

4.6.6 Politiche di pulizia

La paginazione funziona bene quando c'è una gran quantità di pagine fisiche libere che possono essere allocate in seguito ad un fault di pagina; se tutte le pagine fisiche sono piene, e per di più modificate, prima di caricare una nuova pagina bisogna scaricare un'pagina vecchia su disco. Per assicurare un rifornimento abbondante di pagine fisiche libere, molti sistemi di paginazione hanno un processo in background, detto **demone di paginazione**, che è inattivo per la maggior parte del tempo, ma che viene risvegliato ad intervalli fissi per ispezionare lo stato della memoria. Se ci sono troppe poche pagine fisiche libere, il demone di paginazione inizia a selezionare le pagine da scaricare usando l'algoritmo di rimpiazzamento prescelto, e se queste pagine sono state modificate dopo il caricamento, vengono scritte sul disco.

Ad ogni buon conto, viene memorizzato quale era il contenuto precedente delle pagine, e nel caso in cui una delle pagine scaricate venga richiesta prima che sia stata sovrascritta, essa viene semplicemente riallocata togliendola dall'insieme delle pagine libere. Mentre un rifornimento di pagine libere dà prestazioni migliori che allocare tutta la memoria disponibile e poi cercare una pagina da scaricare quando se ne ha la necessità. Il demone di paginazione assicura almeno che tutte le pagine libere siano pulite, così che esse non debbano essere scaricate sul disco in tutta fretta quando vengono richieste.

Un modo per implementare questa tecnica di pulizia è di utilizzare un orologio a due lancette. La prima lancetta è controllata dal demone di paginazione; quando essa punta ad una pagina sporca, la pagina viene scritta su disco e la lancetta avanza; quando punta ad una pagina pulita, avanza semplicemente. La seconda lancetta è utilizzata per il rimpiazzamento delle pagine, come nell'algoritmo standard dell'orologio. Ora, con l'utilizzo del demone di paginazione, la probabilità che la seconda lancetta trovi una pagina pulita è incrementata.

4.6.7 Interfaccia di memoria virtuale

Fino ad ora, nelle nostre discussioni abbiamo ipotizzato che la memoria virtuale sia trasparente ai processi e ai programmati, cioè, tutto quello che essi vedono è un grande spazio di indirizzamento virtuale su un calcolatore con una memoria fisica piccola (più piccola). In alcuni sistemi questo è vero, ma in alcuni sistemi avanzati, i programmati hanno controllo sulla mappa della memoria e possono utilizzarla in modi non tradizionali per migliorare il comportamento del programma. In questa sezione, osserveremo brevemente alcuni di questi utilizzi.

Una ragione per cui si fornisce ai programmati un controllo sulla mappa della memoria è per consentire a due o più processi di condividere la stessa memoria; infatti se i programmati possono attribuire nomi alle regioni della loro memoria, un processo potrà dare ad un altro processo il nome di una regione di memoria, in modo che quest'altro processo possa anch'esso mapparlo. Con due (o più) processi che condividono le stesse pagine, diventa possibile condividere su banda larga - un processo scrive nella memoria condivisa e un altro legge dalla memoria.

La condivisione delle pagine può anche essere utilizzata per implementare un sistema di scambio di messaggi ad alte prestazioni. Normalmente, quando i messaggi vengono scambiati, i dati sono copiati da uno spazio di indirizzamento ad un altro, a costi consi-

derevoli; se i processi possono controllare la mappa delle loro pagine, si può scambiare un messaggio in questo modo: il processo che spedisce il messaggio toglie la mappa alle pagine contenenti il messaggio, ed il processo ricevente invece le mappa. Così, basta copiare soltanto i nomi delle pagine, invece di tutti i dati.

Un'altra tecnica avanzata di gestione della memoria è la **memoria condivisa distribuita** (Feeley ed altri, 1995; Li, 1986; Li e Hudak, 198; e Zekauskas ed altri, 1994), l'idea qui è di consentire a diversi processi in rete di condividere un insieme di pagine, possibilmente, ma non necessariamente, come un singolo spazio di indirizzamento lineare condiviso. Quando un processo riferisce una pagina che non è correntemente mappata, esso genera un fault di pagina, ed il gestore dei fault di pagina, che può essere nel kernel o nello spazio utente, localizza quindi la macchina che contiene le pagine e le spedisce un messaggio chiedendo di togliere la mappa alla pagina e di spedirla sulla rete. Quando la pagina arriva, essa viene mappata e l'istruzione che ha causato il fault viene reinizializzata. Esamineremo con maggior dettaglio la memoria condivisa distribuita nel Capitolo 8.

4.7 Questioni implementative

Gli implementatori dei sistemi di memoria virtuale devono scegliere tra i principali algoritmi teorici, ad esempio fra l'algoritmo della seconda opportunità e quello dell'incremento, tra l'allocazione di pagine di tipo locale o globale, e tra paginazione a richiesta e pre-paginazione; ma devono anche essere a conoscenza di diversi problemi implementativi pratici. In questa sezione daremo un'occhiata ad alcuni dei problemi più comuni e ad alcune loro soluzioni.

4.7.1 Implicazioni della paginazione sul sistema operativo

Vi sono quattro momenti in cui il sistema operativo deve lavorare per la paginazione: durante la creazione dei processi, durante l'esecuzione dei processi, durante i fault di pagina, e durante la terminazione dei processi. Di seguito esamineremo brevemente ciascuno di questi momenti per vedere cosa si deve fare.

Quando in un sistema a paginazione viene creato un nuovo processo, il sistema operativo deve determinare quanto dovranno essere grandi (inizialmente) il programma e i dati, e dovrà creare una tabella delle pagine per essi; deve allocare lo spazio in memoria per la tabella delle pagine, e deve inizializzare questo spazio. Quando il processo viene spostato su disco, la tabella delle pagine non deve essere residente in memoria, ma deve rimanere in memoria quando il processo sta girando; inoltre, deve essere allocato dello spazio su disco per l'area di swap, in modo che quando una pagina viene scaricata, si sappia dove metterla. Anche l'area di swap deve essere inizializzata con il testo del programma e i dati, in modo che quando il nuovo processo inizia a causare fault di pagina, si possono caricare da disco le pagine; alcuni sistemi paginano il testo del programma direttamente dal file eseguibile, quindi risparmiano lo spazio su disco ed il tempo di inizializzazione. Infine, le informazioni riguardo la tabella delle pagine e l'area di swap su disco devono essere memorizzate nella tabella dei processi.

Quando un processo è schedulato per essere eseguito, si deve reimpostare la MMU per il nuovo processo, ed il TLB deve essere svuotato, per eliminare le tracce del processo precedentemente in esecuzione. La nuova tabella delle pagine deve diventare la tabella corrente, generalmente copiandola oppure facendola puntare da qualche registro

hardware; facoltativamente, alcune o tutte le pagine del processo possono essere caricate in memoria per ridurre il numero dei fault di pagina iniziali.

Quando avviene un fault di pagina, il sistema operativo deve leggere i registri hardware per determinare quale indirizzo virtuale ha causato il fault, e da questa informazione esso deve calcolare quale pagina è necessaria e deve localizzare quella pagina su disco; deve quindi trovare una pagina disponibile per inserirvi dentro la nuova pagina, scaricando, se necessario, alcune pagine vecchie; quindi deve leggere le pagine necessarie nella memoria fisica, ed infine deve salvare il contatore di programma, di modo che esso punti all'istruzione che ha generato il fault di pagina, per consentire a quella istruzione di essere eseguita nuovamente.

Quando un processo termina, il sistema operativo deve rilasciare la sua tabella delle pagine, le sue pagine, e lo spazio su disco che le pagine occupavano quando erano su disco; se alcune delle pagine sono condivise con altri processi, le pagine in memoria e su disco possono essere rilasciate solo quando l'ultimo processo che le utilizza è terminato.

4.7.2 Trattamento dei fault di pagina

Siamo finalmente in condizione di descrivere dettagliatamente cosa succede quando si verifica un fault di pagina. La sequenza degli eventi è la seguente:

1. L'hardware provoca una trap al nucleo, salvando il contatore di programma sullo stack. Su molte macchine, le informazioni sullo stato dell'istruzione corrente vengono salvate in registri speciali della CPU.
2. Viene fatta partire una procedura in assembler per salvare i registri generali e le altre informazioni volatili, per evitare che vengano alterate dal sistema operativo. Questo codice chiama il sistema operativo come se fosse una procedura.
3. Il sistema operativo scopre che si è verificato un fault di pagina e cerca di capire qual è la pagina virtuale richiesta. Spesso questa informazione è contenuta in uno dei registri hardware; se non lo è, il sistema operativo deve recuperare il contatore di programma, prelevare l'istruzione ed esaminarla via software per scoprire cosa stava facendo quando si è verificato il fault.
4. Una volta noto l'indirizzo virtuale che ha causato il fault, il sistema operativo controlla che sia un indirizzo valido e che i diritti di protezione siano compatibili con l'accesso. Se non è così, al processo viene spedito un segnale oppure viene ucciso; se l'indirizzo è valido e non vi sono fault di protezione, il sistema tenta di ottenere una pagina fisica dalla lista delle pagine libere. Se non ci sono pagine libere, si fa girare l'algoritmo di rimpiazzamento per selezionare una vittima.
5. Se la pagina selezionata è stata modificata, la si schedula per essere trasferita sul disco e si ha un cambio di contesto, che sospende il processo che aveva generato il fault e ne manda in esecuzione un altro fino a quando il trasferimento sul disco non è stato completato. In ogni caso, la pagina viene marcata come occupata per impedire che essa possa essere usata per qualche altro scopo.
6. Se la pagina non è stata modificata (o immediatamente dopo che sia stata trasferita sul disco) il sistema operativo cerca l'indirizzo sul disco dove si trova la pagina richiesta e schedala una richiesta al disco per caricarla. Mentre la pagina viene caricata, il processo che ha generato il fault è ancora sospeso e viene mandato in esecuzione un altro processo, ammesso che ve ne siano altri.
7. Quando l'interruzione dal disco indica che la pagina è arrivata, le tabelle delle

pagine vengono aggiornate per riportare la sua posizione e la pagina fisica viene marcata come stato normale.

8. L'istruzione che aveva provocato il fault viene ripristinata al punto in cui era cominciata ed il contatore di programma viene ripristinato in modo da puntare a quell'istruzione.
9. Il processo che aveva causato il fault viene schedulato ed il sistema operativo restituisce il controllo alla procedura assembler che lo aveva chiamato.
10. Questa procedura recupera i registri e le altre informazioni volatili e restituisce il controllo allo spazio utente perché l'esecuzione continui, come se non fosse accaduto alcun fault di pagina.

4.7.3 Backup delle istruzioni

Quando un programma fa riferimento ad una pagina che non è in memoria, l'istruzione che provoca il fault viene bloccata a metà esecuzione e si genera una trap al sistema operativo. Dopo che il sistema operativo ha prelevato la pagina richiesta, deve far ripartire l'istruzione che ha provocato la trap e ciò è più facile a dirsi che a farsi.

Per vedere la natura del problema nel suo aspetto più brutto, si consideri una CPU che ha delle istruzioni con due indirizzi, come ad esempio nel Motorola 680x0, molto usato nei sistemi embedded, ad esempio, l'istruzione

MOV.L #6(A1), 2(A0)

è di 6 byte (vedi Figura 4.32). Per far ripartire l'operazione, il sistema operativo deve determinare dove si trova il primo byte dell'istruzione. Il valore del contatore di programma al momento della trap dipende da quale operando ha generato il fault e da come è stato implementato il microcodice della CPU.

In Figura 4.32 abbiamo un'istruzione che inizia all'indirizzo 1000 e che fa tre riferimenti in memoria: la parola dell'istruzione e due offset per gli operandi. A seconda di quali di questi tre riferimenti in memoria abbia provocato il fault di pagina, il contatore di programma al momento del fault potrebbe essere 1000, 1002 o 1004. Molto spesso, è impossibile per il sistema operativo determinare in modo non ambiguo l'inizio delle istruzioni. Se al momento del fault il contatore di programma è 1002, il sistema operativo non ha nessun modo per dire se la parola all'indirizzo 1002 è un indirizzo di memoria associato ad una istruzione di indirizzo 1000 (per esempio la locazione di un operando), o un codice operativo di un'istruzione.

Per quanto questo esempio possa sembrare cattivo, potrebbe anche accadere di peggio. Alcuni modelli di indirizzamento del 680x0 utilizzano l'autoincremento, il che significa che un effetto dell'esecuzione dell'istruzione è quello di incrementare uno o più

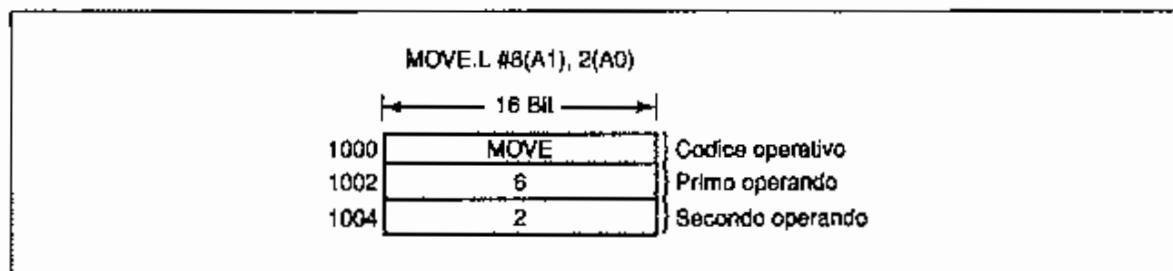


Figura 4.32 Un'istruzione che provoca un fault di pagina.

registri. Anche le istruzioni che usano la modalità di autoincremento, possono generare un fault. A seconda dei dettagli del microcodice, l'incremento può essere fatto prima del riferimento in memoria, e in questo caso il sistema operativo deve decrementare il registro via software prima di far ripartire l'istruzione. Altrimenti, l'autoincremento potrebbe essere effettuato dopo il riferimento in memoria e, in tal caso, non sarebbe ancora stato fatto al momento della trap, ed il sistema operativo non sarebbe obbligato ad annullarlo. L'autodecremento, quando utilizzato, provoca lo stesso problema. I dettagli specifici che stabiliscono se l'autoincremento o autodecremento deve o no deve avvenire prima del corrispondente riferimento alla memoria possono variare da istruzione a istruzione e da un modello di CPU all'altro.

Per fortuna, su alcune macchine, i progettisti della CPU forniscono una soluzione, di solito sotto forma di un registro interno nascosto, in cui viene copiato il contatore di programma subito prima che ogni istruzione venga eseguita. Queste macchine generalmente hanno anche un secondo registro per indicare quali registri sono già stati autoincrementati o autodecrementati e di quanto. Con queste informazioni, il sistema operativo può annullare in maniera non ambigua tutti gli effetti dell'istruzione che ha generato il fault in modo che questa possa essere rieseguita da capo. Se questa informazione non è disponibile, il sistema operativo deve fare salti mortali per cosa accade e come riparare ai danni. È come se i progettisti hardware fossero incapaci di risolvere il problema e ad un certo punto alzassero le mani e dicessero a coloro che scrivono il sistema operativo di pensarci loro. Bravi ragazzi.

4.7.4 Blocco delle pagine in memoria

Sebbene in questo capitolo non si sia parlato molto di ingresso/uscita, il fatto che un calcolatore abbia la memoria virtuale non significa che l'ingresso/uscita sia assente; la memoria virtuale e l'ingresso/uscita possono interagire in maniera molto subdola. Consideriamo un processo che abbia appena generato una chiamata di sistema per leggere qualche cosa da un qualche file o dispositivo in un buffer compreso nel suo spazio di indirizzamento. Mentre sta aspettando che l'ingresso/uscita venga completato, il processo viene sospeso e ne viene mandato in esecuzione un altro, che provoca un fault di pagina.

Se l'algoritmo di paginazione è globale, esiste una possibilità, piccola ma non nulla, che venga scelta la pagina contenente il buffer di ingresso/uscita come pagina da rimuovere; se il dispositivo di ingresso/uscita stesse trasferendo in DMA i dati verso quella pagina, la sua rimozione potrebbe far sì che parte dei dati vengano scritti nel buffer al quale appartengono e parte vengano invece scritti nella pagina appena caricata. Una soluzione a questo problema è quella di bloccare in memoria le pagine coinvolte in operazioni di ingresso/uscita, in maniera che esse non possano essere rimosse. Il blocco di una pagina è spesso detto **pinning** (appuntare con uno spillo) della pagina in memoria. Un'altra soluzione è quella di eseguire tutto l'ingresso/uscita su buffer che stanno nel nucleo, e di copiare i dati nelle pagine dell'utente in un momento successivo.

4.7.5 Memoria secondaria

Nella nostra discussione sugli algoritmi di rimpiazzamento delle pagine abbiamo visto come viene selezionata una pagina per la rimozione, ma non abbiamo detto molto su dove viene posta su disco quando viene scaricata; descriviamo ora alcuni problemi legati alla gestione del disco.

L'algoritmo più semplice per allocare spazio per una pagina sul disco è quello di avere sul disco una speciale area di swap (scambio). Quando si fa partire il sistema, questa

sistema operativo viene trasportato. Il gestore dei fault di pagina è codice indipendente dalla macchina e contiene la maggior parte del meccanismo di paginazione. La politica è determinata in gran parte dal paginatore esterno, che gira come processo utente.

Quando un processo parte, il paginatore esterno viene notificato, in modo da impostare la mappa delle pagine del processo e per allocare, se necessaria, l'area di swap su disco; quando il processo gira, può mappare nuovi oggetti nel suo spazio di indirizzamento, nel qual caso il paginatore esterno verrà nuovamente notificato.

Una volta che il processo inizia a girare, esso può generare un fault di pagina. Il gestore dei fault evidenzia quale pagina virtuale è necessaria e spedisce un messaggio al paginatore esterno, indicandogli qual è il problema; il paginatore esterno, quindi, legge la pagina necessaria dal disco e la copia in una porzione del suo spazio di indirizzamento, quindi, dice al gestore dei fault dov'è la pagina; il gestore dei fault quindi toglie la mappa della pagina relativa allo spazio di indirizzamento del paginatore esterno e chiede al gestore della MMU di mettere la pagina nello spazio di indirizzamento dell'utente al posto corretto; a questo punto il processo utente può ripartire.

Questa implementazione non precisa dove inserire l'algoritmo di rimpiazzamento delle pagine; potrebbe essere più pulito averlo nel paginatore esterno, ma ci sono alcuni problemi con questo approccio; quello principale è che il paginatore esterno non può accedere ai bit *R* ed *M* delle pagine, che giocano un ruolo importante in molti algoritmi di paginazione. Perciò o sono necessari anche meccanismi per passare queste informazioni al paginatore esterno, oppure l'algoritmo di rimpiazzamento delle pagine deve girare nel nucleo. Nell'ultimo caso, il gestore dei fault indica al paginatore esterno quali pagine da scaricare ha selezionato e fornisce i dati, o mappandoli nello spazio di indirizzamento del paginatore esterno oppure includendoli in un messaggio; in entrambi i modi, il paginatore esterno scrive i dati su disco.

Il vantaggio principale di questa implementazione è quello di avere un codice più modulare e maggiormente flessibile; lo svantaggio principale è l'overhead extra che si verifica nell'attraversare i confini tra nucleo ed utente diverse volte, e l'ulteriore overhead dei vari messaggi che devono essere scambiati tra le parti del sistema. Attualmente, questo meccanismo è altamente controverso, ma poiché i calcolatori diventano sempre più veloci, ed il software diventa sempre più complesso, nel lungo periodo molti implementatori accetteranno di sacrificare un po' le prestazioni per avere un software maggiormente affidabile.

4.8 Segmentazione

La memoria virtuale descritta fino ad ora è unidimensionale dal momento che gli indirizzi virtuali variano dall'indirizzo 0 ad un indirizzo massimo, un indirizzo dopo l'altro; per molti problemi, può essere molto meglio avere uno o più spazi di indirizzamento separati piuttosto che averne uno solo. Per esempio, un compilatore ha tante tabelle che vengono costruite man mano che procede la compilazione, fra le quali troviamo solitamente:

1. Il testo sorgente, che viene salvato per la stampa dei listati (nei sistemi batch).
2. La tabella dei simboli, contenente i nomi e gli attributi delle variabili.
3. La tabella che contiene tutte le costanti usate, intere e in virgola mobile.
4. L'albero di derivazione, contenente l'analisi sintattica del programma.
5. Lo stack, usato per le chiamate di procedura del compilatore.

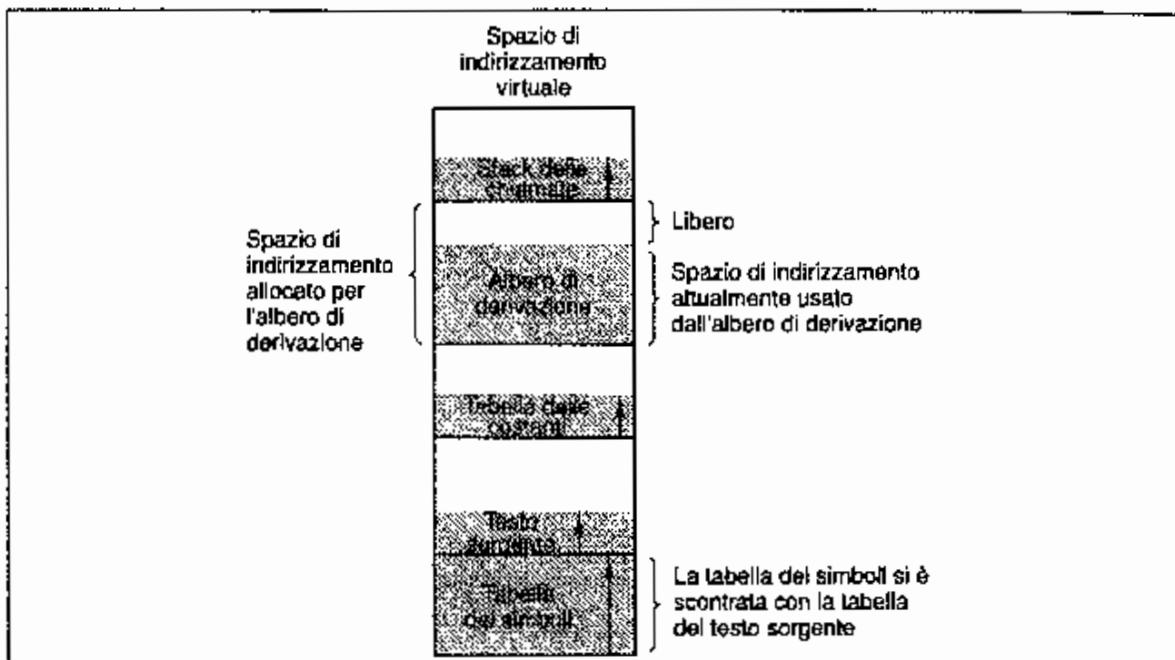


Figura 4.35 In uno spazio di indirizzamento unidimensionale con tabelle che crescono, una tabella può scontrarsi con un'altra.

Ognuna delle prime quattro tabelle cresce man mano che la compilazione va avanti; l'ultima cresce e diminuisce in maniera non prevedibile durante il processo di compilazione. In una memoria unidimensionale, queste cinque tabelle dovrebbero essere allocate in porzioni contigue di memoria, come indicato in Figura 4.35.

Consideriamo cosa accade se un programma ha un numero eccezionalmente alto di variabili, ma un numero normale di tutte le altre cose: la porzione di memoria allocata per la tabella dei simboli potrebbe riempirsi, ma ci potrebbe essere ancora molto spazio libero nelle altre tabelle. Naturalmente, il compilatore potrebbe inviare un messaggio che dica che la compilazione non può continuare per via delle troppe variabili, ma così facendo non si fa una bella figura se nelle altre tabelle esiste spazio inutilizzato.

Un'altra possibilità è quella di giocare a Robin Hood, togliendo spazio alle tabelle che ne hanno molto per darlo a quelle che ne hanno poco. Questo scambio può essere effettuato, ma è come amministrare da soli i propri overlay: nel migliore dei casi è una cosa noiosa, nel peggiore è una considerevole massa di lavoro, noioso ed infruttuoso.

Ciò che serve veramente è un modo per liberare il programmatore della necessità di gestire le tabelle che si espandono e si contraggono, allo stesso modo in cui la memoria virtuale elimina il problema di dover organizzare il programma in overlay.

Una soluzione semplice ed estremamente generale è quella di dotare la macchina di molti spazi di indirizzamento completamente indipendenti, chiamati segmenti. Ciascun segmento consiste di una sequenza lineare di indirizzi, da 0 ad un qualche massimo; la lunghezza di ciascun segmento può andare da 0 al massimo stabilito. Segmenti diversi possono avere lunghezze diverse e di solito così accade; in più le lunghezze dei segmenti possono cambiare durante l'esecuzione. La lunghezza del segmento dello stack può essere allungata quando vi si carica qualcosa e accorciata quando vi si preleva qualcosa' altro.

Dal momento che ciascun segmento costituisce uno spazio di indirizzamento separato, segmenti diversi possono crescere e contrarsi in maniera indipendente senza influenz-

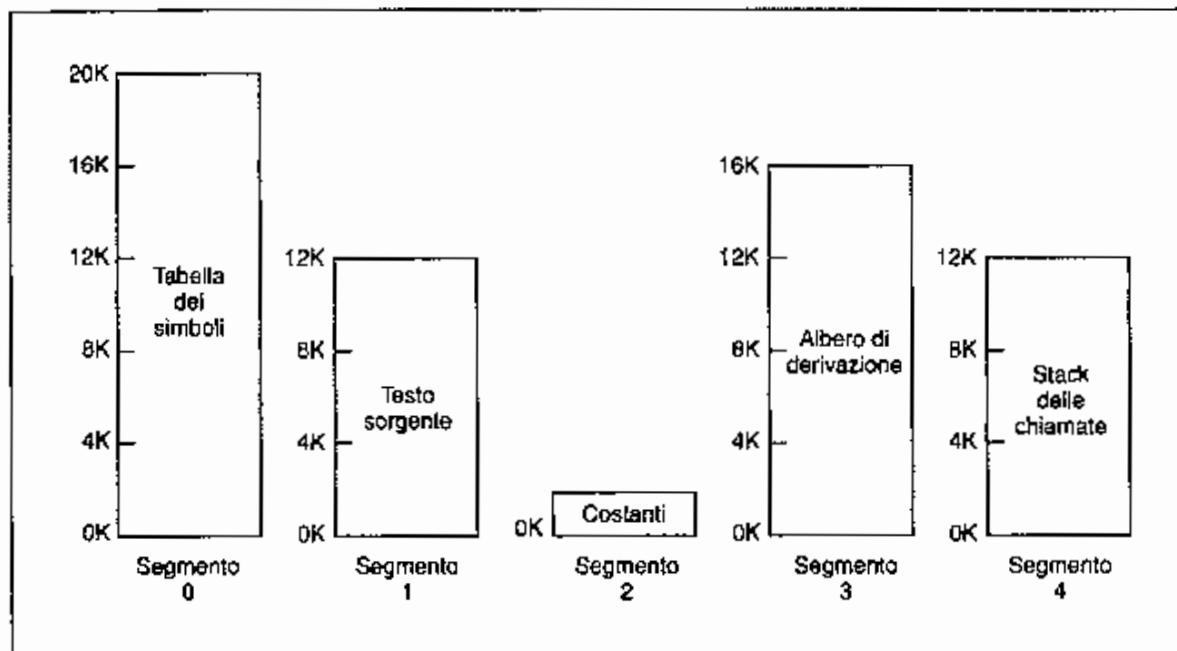


Figura 4.36 Una memoria segmentata permette a ciascuna tabella di allargarsi e contrarsi indipendentemente dalle altre tabelle.

ze reciproche. Se uno stack che stava in un certo segmento ha necessità di crescere, può farlo dal momento che non c'è nient'altro, in quello spazio di indirizzamento, su cui può andare a sbattere. Naturalmente, un segmento si può riempire, ma normalmente i segmenti sono molto grandi e così questo evento è piuttosto raro. Per specificare un indirizzo in questa memoria segmentata o bidimensionale il programma deve fornire un indirizzo formato da due parti: un numero di segmento ed un indirizzo all'interno del segmento. La Figura 4.36 raffigura una memoria segmentata che viene usata per le tabelle del compilatore appena discusse; essa mostra cinque segmenti indipendenti.

Sottolineiamo il fatto che un segmento è un'entità logica, nota al programmatore, e che il programmatore usa effettivamente come una singola entità logica. Un segmento può contenere una procedura, o una matrice, o una pila, o una collezione di variabili scalari, ma di solito non contiene un mixto di tipi diversi.

Una memoria segmentata presenta altri vantaggi, oltre a semplificare la gestione delle strutture dati che crescono o si contraggono; se ciascuna procedura occupa un segmento distinto, con 0 come indirizzo di partenza, il collegamento di procedure compilate separatamente risulta grandemente semplificato. Dopo che tutte le procedure che costituiscono un programma sono state compilate e collegate, una chiamata alla procedura nel segmento n userà l'indirizzo composto da due parti ($n, 0$) per indirizzare la parola 0 (il punto di ingresso).

Se la procedura nel segmento n viene successivamente modificata e ricompilata, non è necessario ricompilare nessun'altra procedura (perché non viene modificato nessun indirizzo di partenza), anche nel caso in cui la nuova versione sia più grande della precedente. Con una memoria unidimensionale, le procedure sono impacchettate una a ridosso dell'altra, senza alcuno spazio di indirizzamento libero fra di loro. Di conseguenza, cambiare la dimensione di una procedura può provocare modifiche nell'indirizzo di altre procedure, anche se non correlate. A sua volta, queste modifiche possono rendere necessarie modifiche a tutte le procedure che chiamano una delle procedure sposta-

Considerazione	Paginazione	Segmentazione
Il programmatore deve sapere che viene usata questa tecnica?	No	Si
Quanti spazi di indirizzamento lineari vi sono?	1	Molti
Lo spazio totale di indirizzamento può eccedere la dimensione della memoria fisica?	Si	Si
Le procedure e i dati possono essere distinti e protetti separatamente?	No	Si
Le tabelle a dimensione variabile possono essere gestite semplicemente?	No	Si
Viene facilitata la condivisione delle procedure fra gli utenti?	No	Si
Perché sono state inventate queste tecniche?	Per ottenere uno spazio di indirizzamento lineare molto grande senza dover comprare altra memoria fisica.	Per permettere di dividere programmi e dati in spazi di indirizzamento indipendenti e per aiutare protezione e condivisione.

Figura 4.37 Confronto fra la paginazione e la segmentazione.

te, al fine di incorporare il nuovo indirizzo di partenza. Se il programma contiene centinaia di procedure, questa operazione può risultare molto costosa.

La segmentazione facilita anche la condivisione delle procedure o dei dati fra i vari processi. Un esempio comune è la libreria condivisa. Le moderne workstation che operano con ambienti a finestre avanzati hanno spesso grandi librerie grafiche compilate praticamente in tutti i programmi; in un sistema con segmentazione, la libreria grafica può essere posta in un segmento e condivisa da tutti i processi, eliminando la necessità di averla nello spazio di indirizzamento di ciascuno dei processi. Pur essendo possibile avere librerie condivise anche nei sistemi con paginazione pura, è molto più complicato; in effetti, questi sistemi realizzano questo meccanismo usando un'emulazione della segmentazione.

Dal momento che ciascun segmento costituisce un'entità logica che è nota al programmatore, per esempio una procedura, una matrice o uno stack, segmenti diversi possono avere protezioni di tipo diverso: un segmento contenente una procedura può essere specificato con diritti di sola esecuzione, proibendovi letture e scritture; una matrice di valori in virgola mobile può essere specificata con i diritti di lettura e scrittura ma non di esecuzione, e si possono intercettare i tentativi di saltare ad indirizzi al suo interno. Questo tipo di protezione è utile per scoprire gli errori di programmazione.

Dovete cercare di capire perché la protezione abbia senso nel caso di una memoria segmentata, ma non nel caso di una memoria unidimensionale paginata. Nella memoria segmentata, l'utente è consci di che cosa vi sia all'interno di ciascun segmento: normalmente, un segmento non conterrà, ad esempio, una procedura ed uno stack, ma l'uno o l'altro. Dal momento che ogni segmento contiene un unico tipo di oggetto, ogni segmento

to può avere la protezione appropriata per l'oggetto che contiene. La paginazione e la segmentazione vengono confrontate in Figura 4.37.

I contenuti di una pagina sono, in un certo senso, casuali; il programmatore non è nemmeno consapevole dell'esistenza della paginazione. Sebbene sia possibile mettere in ciascun elemento della tabella delle pagine alcuni bit che specificano gli accessi permessi ad una pagina, per utilizzare questa caratteristica il programmatore dovrebbe tenere traccia della posizione nello spazio degli indirizzi dove vanno a cadere gli estremi delle pagine, ma questo è precisamente il tipo di compiti che la paginazione intendeva eliminare. Dal momento che l'utente di una memoria segmentata ha l'illusione che tutti i segmenti siano in memoria contemporaneamente, cioè può indirizzarli come se lo fossero, egli può proteggere ciascun segmento separatamente, senza dover farsi carico della loro gestione tramite overlay.

4.8.1 Implementazione della segmentazione pura

L'implementazione della segmentazione differisce da quella della paginazione in un punto fondamentale: le pagine hanno dimensione fissa, mentre i segmenti no. La Figura 4.38(a) mostra un esempio di memoria fisica che inizialmente contiene cinque segmenti. Ora consideriamo cosa accade quando il segmento 1 viene scaricato ed al suo posto viene caricato il segmento 7, più piccolo; si arriva alla configurazione di memoria della Figura 4.38(b): fra il segmento 7 ed il segmento 2 troviamo un'area non usata, cioè un buco. In seguito, il segmento 4 viene sostituito con il segmento 5, come in Figura 4.38(c), ed il segmento 3 con il 6, come in Figura 4.38(d). Dopo che il sistema avrà lavorato per un po', la memoria sarà divisa in un certo numero di porzioni, alcune contenenti segmenti, altre contenenti buchi. Questo fenomeno, detto **checkerboarding** o **frammentazione esterna**, spreca memoria nei buchi. Vi si può porre rimedio con la compattazione, come mostrato in Figura 4.38(e).

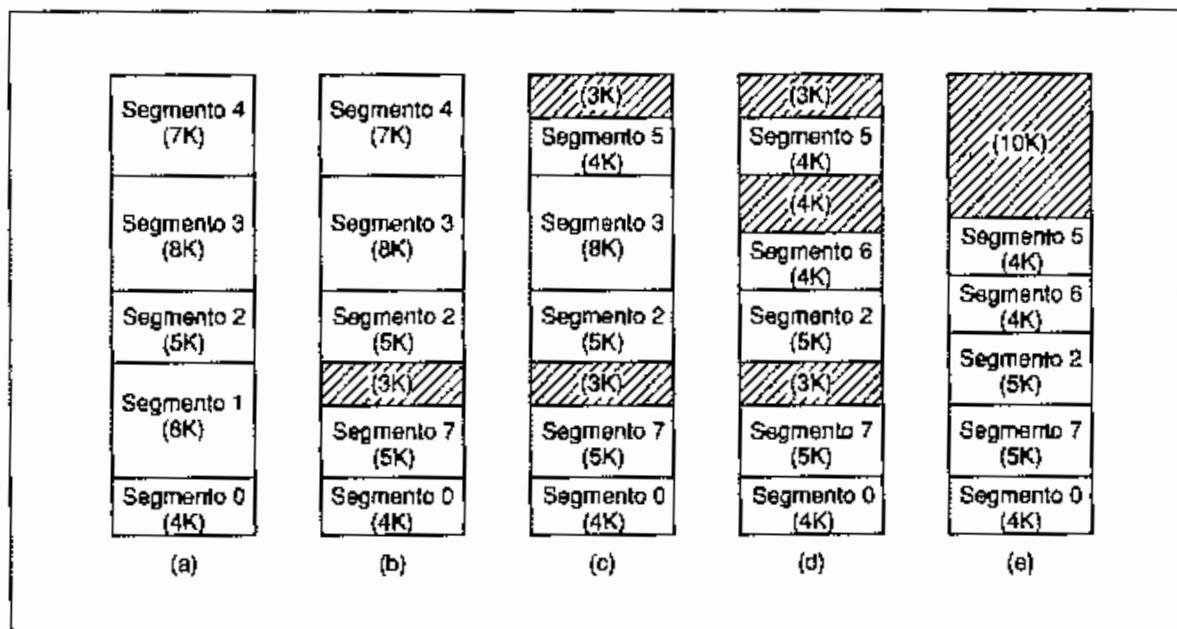


Figura 4.38 (b)-(d) Sviluppo del fenomeno di checkerboarding. (e) Sua rimozione tramite la compattazione.

4.8.2 Segmentazione con paginazione: MULTICS

Se i segmenti sono grandi può risultare poco conveniente, se non impossibile, caricarli in memoria nella loro interezza; questo fatto ha portato all'idea di paginarli, in maniera tale che debbano essere presenti solo le pagine effettivamente necessarie. Due sistemi significativi hanno supportato segmenti paginati: in questa sezione descriveremo il primo di essi, MULTICS, nella prossima discuteremo il secondo, Intel Pentium.

MULTICS gira sulla famiglia Honeywell 6000 e sui suoi discendenti e mette a disposizione di ciascun programma una memoria virtuale formata da un massimo di 2^{18} segmenti (più di 250000) ciascuno dei quali può essere lungo fino a 65536 parole da 36 bit. Per implementare questo schema, i progettisti di MULTICS scelsero di trattare ogni segmento come una memoria virtuale e, di conseguenza, di paginarlo, combinando i vantaggi della paginazione (dimensione uniforme delle pagine e non dover mantenere l'intero segmento in memoria, ma solo la parte effettivamente necessaria) con i vantaggi della segmentazione (facilità di programmazione, modularità, protezione e condivisione).

Ciascun processo MULTICS ha una tabella dei segmenti, con un descrittore per segmento. Dal momento che potenzialmente può esistere oltre un quarto di milione di segmenti, la tabella dei segmenti è essa stessa un segmento, gestito tramite paginazione. Un descrittore di segmento indica se il segmento è in memoria o no; se il segmento è almeno parzialmente in memoria, la sua tabella delle pagine sarà anch'essa in memoria, ed il suo descrittore conterrà un puntatore di 18 bit alla sua tabella delle pagine [vedere Figura 4.39(a)]. Dal momento che gli indirizzi fisici sono da 24 bit e che tutte le pagine sono allineate a multipli di 64 byte (cioè i 6 bit di ordine più basso di ogni indirizzo di pagina valgono 000000), sono necessari solo 18 bit nel descrittore per memorizzare un indirizzo di tabella delle pagine. Il descrittore contiene anche la dimensione del segmento, i bit di protezione e alcuni altri elementi. La Figura 4.39(b) illustra un descrittore di segmento di MULTICS. L'indirizzo del segmento nella memoria secondaria non sta nel descrittore del segmento, ma in un'altra tabella usata dal gestore dei fault del segmento.

Ciascun segmento rappresenta un normale spazio di indirizzamento virtuale e viene gestito tramite paginazione allo stesso modo in cui viene gestita la memoria paginata e non segmentata descritta precedentemente in questo capitolo. La dimensione normale delle pagine è di 1024 parole (ma alcuni segmenti piccoli usati dallo stesso MULTICS non sono paginati o vengono paginati in unità da 64 parole, per risparmiare memoria).

Un indirizzo in MULTICS si compone di due parti: il segmento e l'indirizzo all'interno del segmento, quest'ultimo ulteriormente diviso in: numero di pagina e parola all'interno della pagina, come illustrato in Figura 4.40. Quando si verifica un riferimento alla memoria, si esegue il seguente algoritmo:

1. Il numero di segmento viene usato per reperire il descrittore del segmento.
2. Si esegue un controllo per verificare se la tabella delle pagine relativa al segmento sta in memoria. Se è in memoria la si localizza, altrimenti si genera un fault di segmento. Se si verifica una violazione di protezione si genera un fault (trap).
3. Si esamina l'elemento della tabella delle pagine relativo alla pagina virtuale richiesta. Se la pagina non è in memoria, si genera un fault di pagina. Se lo è, dall'elemento della tabella delle pagine si estrae l'indirizzo di memoria principale di inizio della pagina.
4. Si somma l'offset all'indirizzo di partenza della pagina per generare l'indirizzo fisico di memoria dove si trova la parola.
5. Infine, viene effettuata la lettura o la scrittura.

Il processo è illustrato in Figura 4.41. Per semplicità, è stato omesso il fatto che il seg-

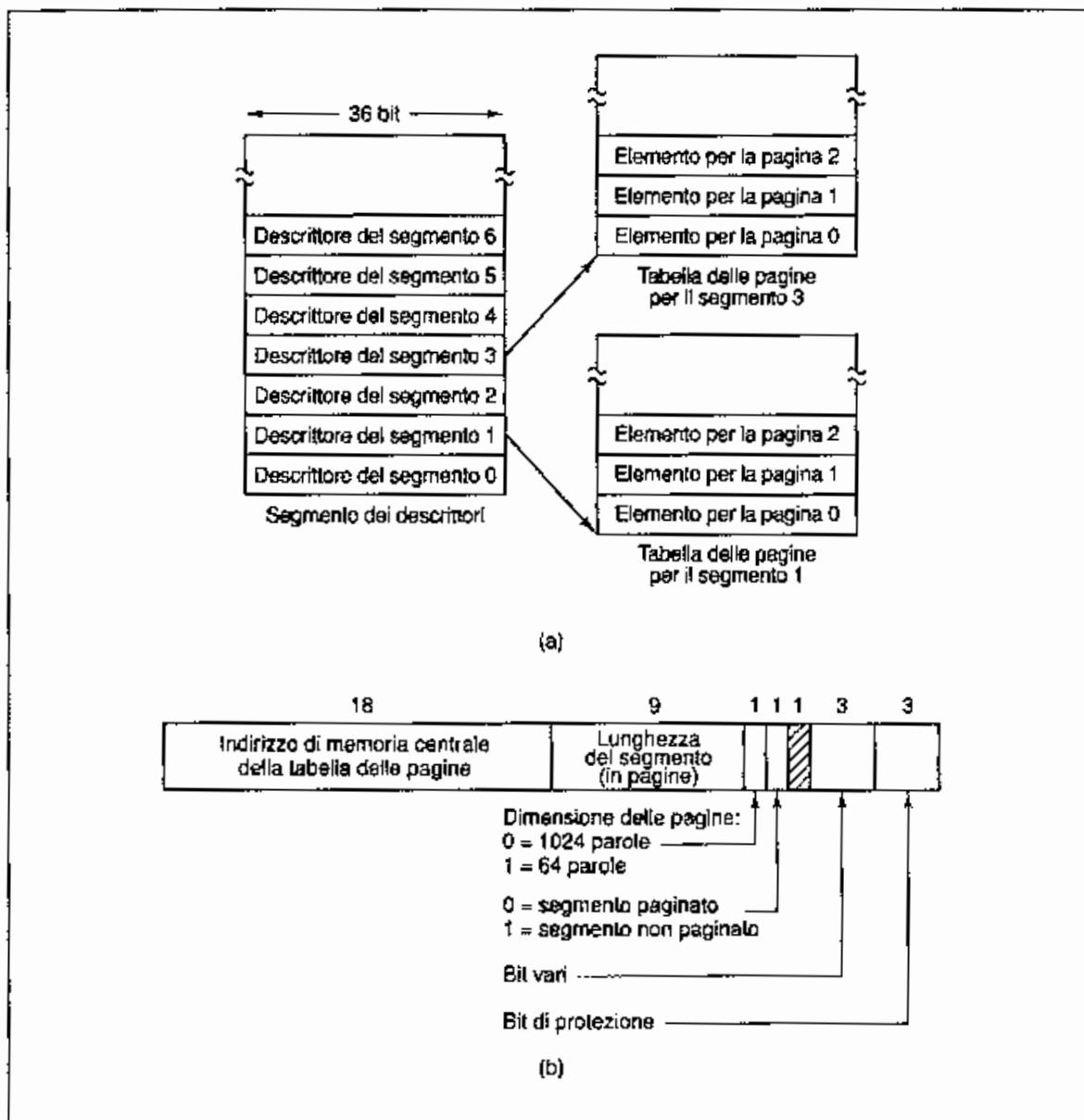


Figura 4.39 La memoria virtuale di MULTICS. (a) Il segmento dei descrittori punta alle tabelle delle pagine. (b) Un descrittore di segmento. I numeri rappresentano la lunghezza dei campi.

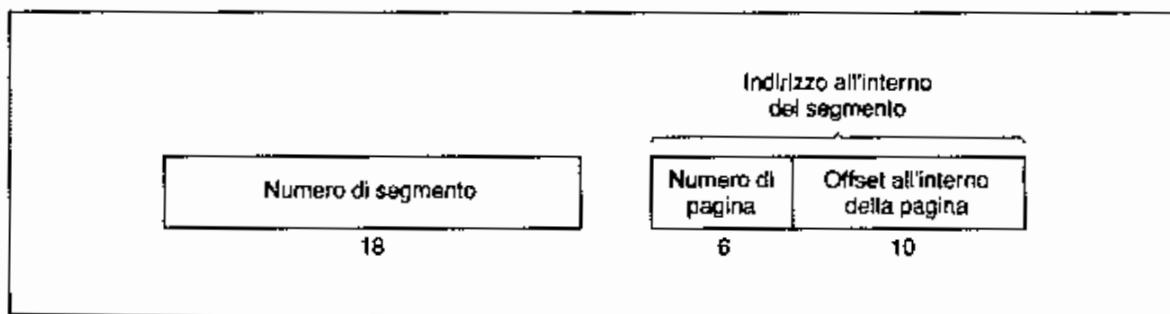


Figura 4.40 Un indirizzo virtuale di MULTICS da 34 bit.

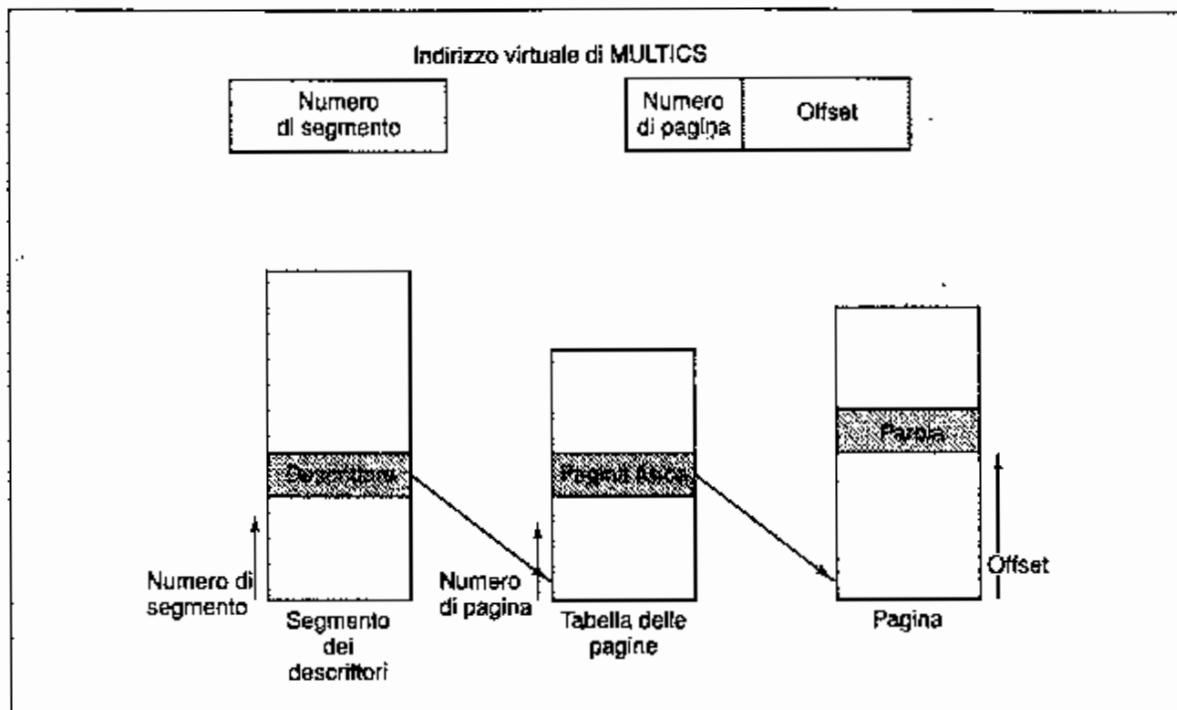


Figura 4.41 Conversione di un indirizzo MULTICS a due componenti in un indirizzo di memoria principale.

mento dei descrittori sia esso stesso paginato. Ciò che succede realmente è che viene usato un registro (il registro base del descrittore) per individuare la tabella delle pagine del segmento dei descrittori che, a sua volta, punta alle pagine del segmento dei descrittori. Una volta trovato il descrittore del segmento desiderato, l'indirizzamento prosegue come mostrato in Figura 4.41.

Come avrete sicuramente immaginato, se l'algoritmo precedente fosse veramente ese-

Campi per il confronto						Questo elemento è stato usato?
Numero di segmento	Pagina virtuale	Pagina fisica	Protezione	Età		
4	1	7	Lettura/Scrittura	13	1	
6	0	2	Sola lettura	10	1	
12	3	1	Lettura/Scrittura	2	1	
					0	
2	1	D	Sola esecuzione	7	1	
2	2	12	Sola esecuzione	9	1	

Figura 4.42 Una versione semplificata del TLB di MULTICS. L'esistenza di pagine di due dimensioni diverse rende più complicato il vero TLB.

guito dal sistema operativo ad ogni istruzione, i programmi non girerebbero molto velocemente. Nella realtà, l'hardware di MULTICS contiene un TLB ad alta velocità di 16 parole che può eseguire una ricerca su una certa chiave in parallelo su tutti i suoi elementi; essa è illustrata in Figura 4.42. Quando il calcolatore genera un indirizzo, l'hardware per l'indirizzamento dapprima controlla se l'indirizzo virtuale sta nel TLB; se c'è prende il numero di pagina fisica direttamente dal TLB e forma l'indirizzo effettivo della parola a cui si deve accedere senza bisogno di cercare nel segmento dei descrittori o nella tabella delle pagine.

Nel TLB vengono mantenuti gli indirizzi delle 16 pagine usate più di recente. I programmi in cui l'insieme di lavoro è più piccolo della dimensione del TLB arriveranno all'equilibrio con gli indirizzi di tutto l'insieme di lavoro nel TLB e, di conseguenza, gireranno in maniera efficiente. Se la pagina non è nel TLB, si esegue un accesso vero e proprio al descrittore ed alla tabella delle pagine per trovare l'indirizzo di pagina fisica e si aggiorna il TLB includendovi questa pagina, scaricando dal TLB la pagina usata meno di recente. Il campo Età tiene traccia della pagina usata meno di recente. La ragione per cui si usa un TLB è la velocità, dal momento che il numero di segmento e di pagina possono essere confrontati in parallelo con quelli di tutti gli elementi.

4.8.3 Segmentazione con paginazione: Pentium Intel

La memoria virtuale del Pentium assomiglia per molti aspetti a quella di MULTICS, dal momento che include sia la segmentazione sia la paginazione. Mentre MULTICS ha 256K segmenti indipendenti, ciascuno di 64K parole da 36 bit, il Pentium ha 16K segmenti indipendenti, ciascuno dei quali può contenere fino a 1000 miliardi di parole da 32 bit. Sebbene ci siano meno segmenti, la dimensione più grossa dei segmenti è di gran lunga più importante, dal momento che pochi programmi richiedono più di 1000 segmenti, ma molti programmi richiedono segmenti grandi.

Il cuore della memoria virtuale del Pentium è dato da due tabelle: la LDT (Local Descriptor Table, tabella dei descrittori locali) e la GDT (Global Descriptor Table, tabella dei descrittori globali). Ciascun programma ha la sua LDT, ma esiste una sola GDT, condivisa fra tutti i programmi del calcolatore. Le LDT descrivono i segmenti locali a ciascun programma, compreso il codice, lo stack, i dati, eccetera, mentre la GDT descrive i segmenti di sistema, compreso lo stesso sistema operativo.

Per accedere ad un segmento, un programma Pentium carica dapprima un selettori di quel segmento in uno dei sei registri di segmento della macchina. Durante l'esecuzione, il registro CS contiene il selettore per il segmento del codice ed il registro DS quello per il segmento dei dati. Gli altri registri di segmento sono meno importanti. Ciascun selettore è un numero da 16 bit, come mostrato in Figura 4.43.

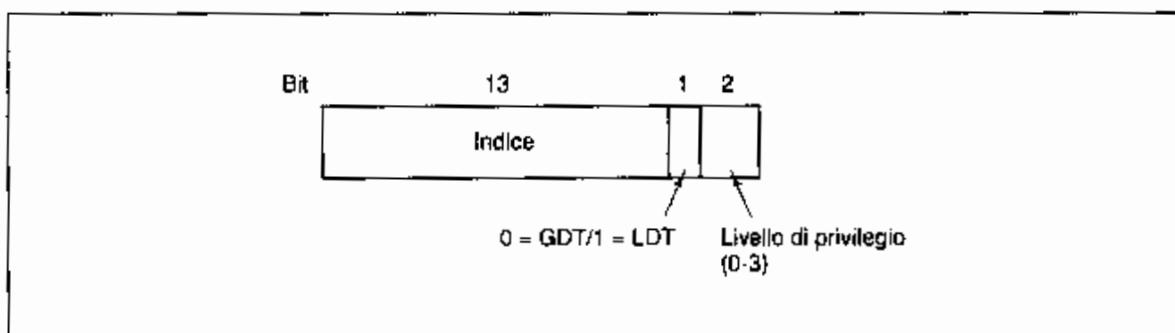


Figura 4.43 Un selettore del Pentium.

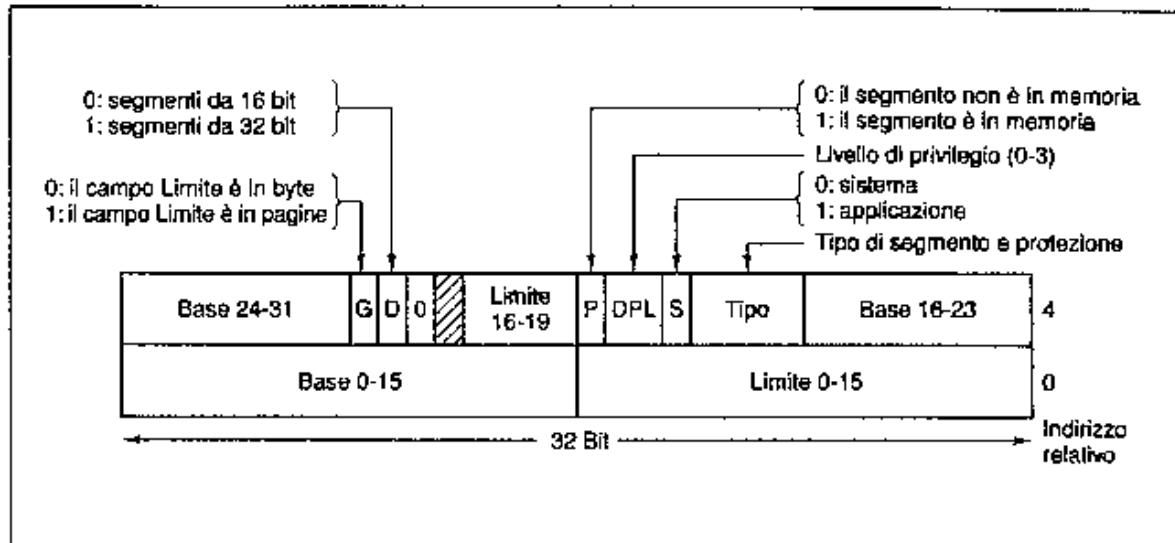


Figura 4.44 Descrittore del segmento di codice Pentium. I descrittori dei segmenti dati sono leggermente diversi.

Uno dei bit del selettor dice se il segmento è locale o globale (cioè se nella LDT o nella GDT). Tredici altri bit specificano il numero di elemento della LDT o della GDT, così che ciascuna di queste tabelle contiene al più 8K descrittori di segmento. Gli altri due bit hanno a che fare con la protezione e verranno descritti più tardi. Il descrittore 0 è proibito: può essere caricato in un registro di segmento solo per indicare che quel registro di segmento non è disponibile in quel momento. Se usato, causa una trap.

Nel momento in cui un selettor viene caricato in un registro di segmento, il corrispondente descrittore viene prelevato dalla LDT o dalla GDT e memorizzato in un registro di microprogramma, in modo che vi si possa accedere velocemente. Un descrittore è formato da 8 byte che includono l'indirizzo di base del segmento, la dimensione e altre informazioni, come illustrato in Figura 4.44.

Il formato del selettor è stato scelto in maniera intelligente per facilitare la localizzazione del descrittore. Prima si seleziona la LDT o la GDT a seconda del contenuto del bit 2 del selettor, poi si copia il selettor in un registro tampone di microprogramma, e i 3 bit di ordine inferiore vengono messi a 0; infine, vi si somma l'indirizzo della tabella LDT o della GDT per ottenere un puntatore diretto al descrittore. Per esempio, il selettor 72 fa riferimento all'elemento 9 della GDT, che si trova all'indirizzo GDT + 72.

Vediamo come, passo passo, una coppia (selettor, offset) venga convertita in un indirizzo di memoria fisica. Non appena il microprogramma sa quale registro di segmento verrà usato, è in grado di trovare il descrittore completo che corrisponde a quel settore nei suoi registri interni; se il segmento non esiste (selettor 0) o in quel momento risulta scaricato, si verifica una trap.

In seguito controlla se l'offset va a cadere oltre i limiti del segmento, ed anche in questo caso viene generata una trap. Dal punto di vista logico, nel descrittore ci dovrebbe essere un campo da 32 bit che indica la dimensione del segmento, ma, dal momento che sono disponibili solo 20 bit, si usa uno schema diverso. Se il campo Gbit (Granularità) è a 0, il campo Limite rappresenta la dimensione esatta del segmento, fino ad un massimo di 1M. Se il campo è 1, il campo Limite dà la dimensione del segmento in pagine, anziché in byte. La pagina del Pentium è fissata a 4KB, così bastano 20 bit per segmenti che arrivano fino a 2^{31} byte.

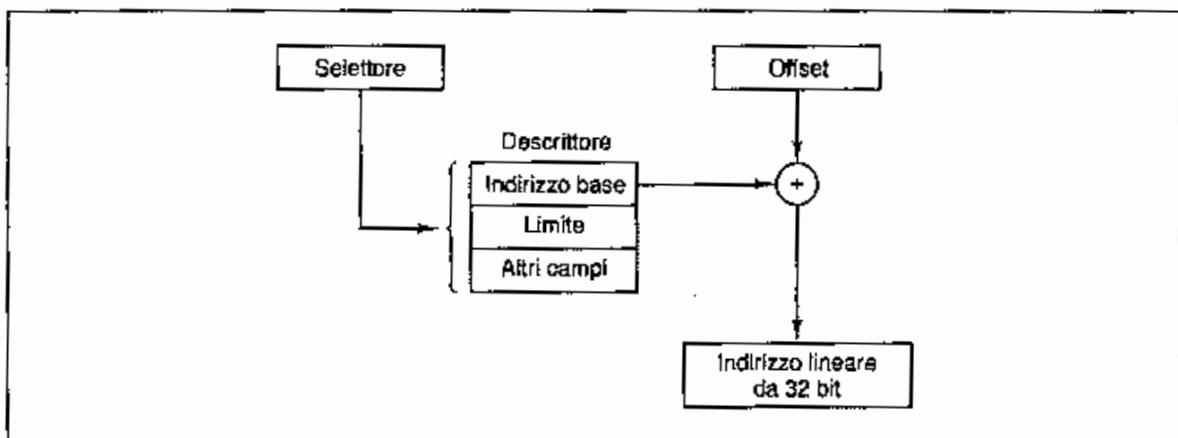


Figura 4.45 Conversione di una coppia {selettores, offset} in un indirizzo lineare.

Supponendo che il segmento sia in memoria e che l'offset cada nell'intervallo dei valori ammessi, il Pentium aggiunge poi all'offset il campo (da 32 bit) *Base* del descrittore, per formare quello che viene chiamato **indirizzo lineare**, come mostrato in Figura 4.45. Il campo *Base* viene diviso in tre pezzi sparpagliati per il descrittore per compatibilità con il 286, nel quale *Base* è di soli 24 bit; in effetti, il campo *Base* fa sì che ogni segmento possa iniziare ad un indirizzo arbitrario all'interno dello spazio di indirizzamento lineare da 32 bit.

Se la paginazione è disabilitata (tramite un bit che si trova in un registro globale di controllo), l'indirizzo lineare viene interpretato come **indirizzo fisico** e viene direttamente

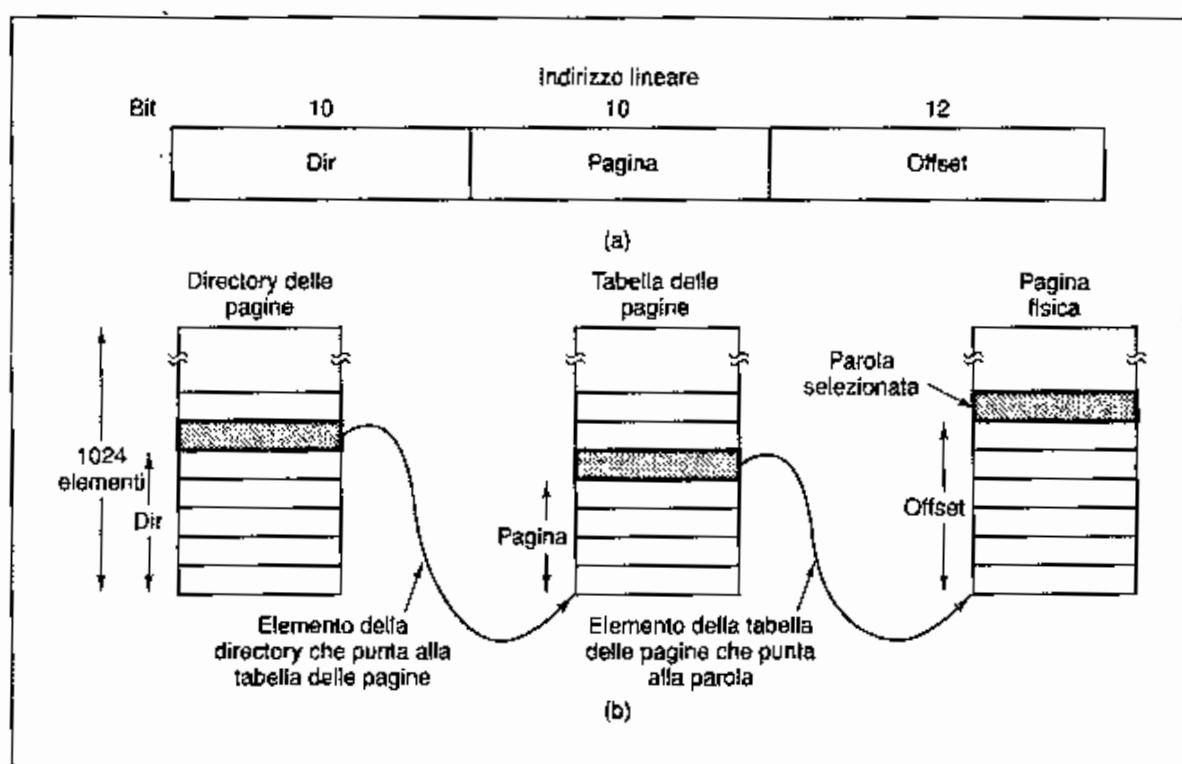


Figura 4.46 Traduzione di un indirizzo lineare in un indirizzo fisico.

te spedito alla memoria per la lettura o la scrittura. Così, con la paginazione disabilitata abbiamo uno schema di segmentazione pura, con l'indirizzo di base di ciascun segmento contenuto nel suo descrittore. Incidentalmente, è permessa la sovrapposizione fra segmenti, probabilmente perché controllare che i segmenti siano disgiunti sarebbe troppo complicato e porterebbe via troppo tempo.

D'altra parte, se la paginazione risulta abilitata, l'indirizzo lineare viene interpretato come un indirizzo virtuale e tradotto in indirizzo fisico usando le tabelle delle pagine in maniera del tutto simile a quanto visto negli esempi precedenti. La sola complicazione è che con un indirizzo virtuale da 32 bit e una pagina da 4KB, un segmento può contenere un milione di pagine, per cui si usa uno schema di traduzione a due livelli per ridurre la dimensione delle tabelle delle pagine per segmenti di piccole dimensioni.

Ciascun programma in esecuzione ha una directory (indice) delle pagine che consta di 1024 elementi da 32 bit, posta ad un indirizzo contenuto in un registro globale. Ciascun elemento in questa directory punta ad una tabella delle pagine, anch'essa contenente 1024 elementi da 32 bit. Gli elementi della tabella delle pagine puntano alle pagine fisiche. Lo schema è illustrato in Figura 4.46.

In Figura 4.46(a) vediamo un indirizzo lineare diviso in tre parti, *Dir*, *Pagina* e *Offset*. Il campo *Dir* viene usato come indice nella directory delle pagine per individuare un puntatore alla tabella delle pagine opportuna; poi si usa il campo *Pagina* come indice per accedere alla tabella delle pagine per trovare l'indirizzo della pagina fisica, infine, il campo *Offset* viene sommato all'indirizzo della pagina fisica per ottenere l'indirizzo della parola o byte che è stata indirizzata.

Gli elementi della tabella delle pagine sono da 32 bit ciascuno, di cui 20 bit contengono il numero della pagina fisica. I rimanenti bit contengono i bit che indicano se la pagina è stata modificata o semplicemente usata, mantenuti dall'hardware a vantaggio del sistema operativo; i bit di protezione ed altri bit di utilità.

Ciascuna delle tabelle delle pagine ha elementi per 1024 pagine fisiche da 4KB, così ogni tabella delle pagine può indirizzare fino a 4MB di memoria. Un segmento più piccolo di 4MB avrà una directory delle pagine con un singolo elemento, il puntatore alla sua unica tabella delle pagine; in questo modo, l'overhead per i segmenti piccoli è di sole due pagine, contro il milione di pagine che sarebbero state necessarie nel caso di tabella delle pagine ad un solo livello.

Per evitare di eseguire continui riferimenti alla memoria, il Pentium, come MULTICS, ha un piccolo TLB che traduce direttamente le combinazioni *Dir-Pagina* usate più di recente in indirizzi di pagina fisica. Solo quando la combinazione richiesta non è presente nel TLB si usa veramente il meccanismo descritto dalla Figura 4.46, e si aggiorna il TLB: se le miss su TLB sono rare, le prestazioni sono buone.

Vale la pena di notare che se un'applicazione particolare non necessita di segmentazione, ma si accontenta di uno spazio di indirizzamento singolo paginato a 32 bit, ebbe ne, questo è un modello possibile. Tutti i registri di segmento possono essere caricati con uno stesso selettor, il cui descrittore ha *Base* = 0 e *Limite* caricato con il valore massimo. L'offset dell'istruzione sarà poi l'indirizzo lineare, e vi sarà un solo spazio di indirizzi in uso, come accade effettivamente con la paginazione normale. Infatti, ad oggi tutti i sistemi operativi per il Pentium funzionano in questo modo, OS/2 era l'unico che utilizzava l'intera potenza dell'architettura MMU dell'Intel.

Dopo tutto, bisogna riconoscere i meriti dei progettisti del Pentium: dati i molti obiettivi in conflitto fra di loro, di implementare la paginazione pura, la segmentazione pura e paginata, mantenendo la compatibilità con il 286, e di fare tutto questo in maniera efficiente, il progetto che ne è risultato è sorprendentemente semplice e pulito.

Sebbene sia stata discussa tutta l'architettura della memoria virtuale del Pentium, anche se in maniera concisa, vale la pena di dire due parole sulla protezione, dal momen-

to che questo argomento è intimamente correlato con la memoria virtuale. Come per lo schema della memoria virtuale, anche lo schema di protezione è strettamente modellato su quello di MULTICS. Il Pentium supporta quattro livelli di protezione, con il livello 0 che rappresenta quello con maggiori privilegi e il livello 3 quello con meno. Questi livelli sono mostrati in Figura 4.47: ad ogni istante, un programma in esecuzione sta ad un certo livello, indicato da un campo a 2 bit della sua PSW. Anche ciascun segmento nel sistema ha un suo livello.

Finché un programma si limita ad usare segmenti del proprio livello, tutto funziona bene. Sono anche permessi tentativi di accedere a dati di più alto livello, ma tentativi di accedere a dati di livello più bassi sono invece illegali e provocano delle trap. I tentativi di chiamare procedure di un livello diverso (più alto o più basso) sono permessi, ma in un modo controllato con attenzione. Per realizzare una chiamata ad un livello diverso, l'istruzione di CALL deve contenere un selettore, invece di un indirizzo. Questo selettore specifica un descrittore detto *call gate* (porta di chiamata) che dà l'indirizzo della procedura da chiamare: così non è possibile saltare nel bel mezzo di un segmento di codice arbitrario di livello diverso, e si possono usare solo i punti di accesso (entry point) ufficiali. I concetti di livelli di protezione e di porte di chiamata furono sperimentati per la prima volta in MULTICS, dove venivano visti come anelli di protezione.

La Figura 4.47 suggerisce un uso tipico di questo meccanismo: a livello 0 troviamo il nucleo del sistema operativo, che gestisce l'ingresso/uscita, si occupa della gestione della memoria e di altre questioni critiche. Al livello 1 si trova il gestore delle chiamate di sistema: i programmi utente possono chiamare procedure che si trovano a questo livello per richiedere l'esecuzione delle chiamate di sistema, ma si possono chiamare solo le procedure specifiche che stanno in una lista protetta di procedure. Il livello 2 contiene le procedure di libreria, possibilmente condivise fra molti programmi che si trovano in esecuzione. I programmi utente possono chiamare queste procedure e leggere i loro dati, ma non li possono modificare. Infine, i programmi utente si trovano al livello 3, che ha la protezione minore.

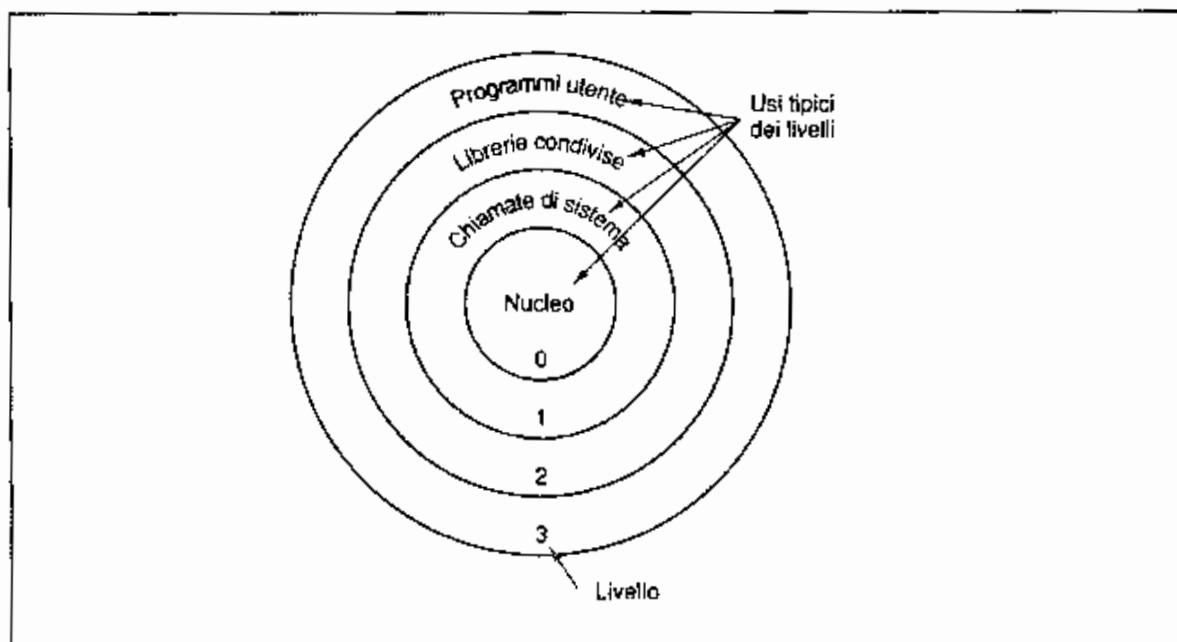


Figura 4.47 Protezione sul Pentium.

Le trap e le interruzioni usano un meccanismo simile alle porte di chiamata: anche loro fanno riferimento a descrittori, piuttosto che ad indirizzi assoluti, e questi descrittori puntano a specifiche procedure da eseguire. Il campo *Type* della Figura 4.44 distingue i segmenti fra segmenti di codice, dati e vari tipi di porte.

4.9 Ricerca sulla gestione della memoria

Una volta la gestione della memoria, specialmente gli algoritmi di paginazione, era terreno fruttifero per le ricerche, ma con il tempo tutto è cambiato, almeno nei sistemi di uso generale. La maggior parte dei sistemi reali tendono ad usare alcune delle variazioni dell'algoritmo dell'orologio, poiché esso è facile da implementare e relativamente efficace. Una recente eccezione, tuttavia, è data dalla riprogettazione del sistema a memoria virtuale 4.4 BSD (Cranor e Parulkar, 1999).

Vi è una ricerca che si sta sviluppando attualmente sulla paginazione nei sistemi specializzati e su alcuni nuovi tipi di sistemi. Parte di questo lavoro viene realizzato per consentire ai processi utente di risolvere i propri fault di pagina e di occuparsi della gestione della propria memoria, possibilmente secondo modalità specifiche per le varie applicazioni (Engler e altri, 1995). Un'area in cui le applicazioni possono avere bisogno di eseguire la propria paginazione secondo modalità speciali è il multimediale, per questo motivo alcune ricerche si occupano di questo obiettivo (Hard, 1999). Un'altra area che ha richieste speciali è la gestione ed il mantenimento dei comunicatori personali palmari (Abutaleb e Li, 1997; e Wan e Lin, 1997). Infine, un'altra area di ricerca riguarda i sistemi con spazi di indirizzamento a 64 bit condivisi da molti processi (Talluri ed altri, 1995).

4.10 Sommario

In questo capitolo abbiamo esaminato la gestione della memoria. Abbiamo visto che i sistemi più semplici non usano per niente lo swapping né la paginazione: una volta che il programma viene caricato in memoria, vi rimane finché non è terminato. Alcuni sistemi operativi permettono di avere un solo processo in memoria alla volta, mentre altri supportano la multiprogrammazione.

Il passo successivo è lo swapping. Quando si usa lo swapping, il sistema è in grado di gestire più processi di quelli per cui ha effettivamente a disposizione spazio in memoria. I processi per i quali non c'è spazio vengono scaricati sul disco. Lo spazio libero in memoria e sul disco viene memorizzato usando delle mappe di bit o liste di buchi.

I calcolatori più avanzati hanno spesso una qualche forma di memoria virtuale. Nella sua forma più semplice, lo spazio di indirizzamento di ciascun processo viene diviso in blocchi di dimensione costante, detti pagine, che possono essere allocati in una qualunque delle pagine fisiche disponibili in memoria. Vi sono molti algoritmi di rimpiazzamento delle pagine; due dei migliori algoritmi sono l'algoritmo dell'invecchiamento e l'algoritmo WSClock.

I sistemi di paginazione possono essere modellati astraendo la stringa di riferimento delle pagine dal programma, ed utilizzando la stessa stringa di riferimento con diversi algoritmi; questi modelli possono essere utilizzati per fare previsioni sul comportamento della paginazione.

Per rendere funzionanti i sistemi di paginazione, non basta scegliere un algoritmo, sono necessarie attenzioni particolari per alcuni concetti come ad esempio la determinazione dell'insieme di lavoro, le politiche di allocazione della memoria, e la dimensione delle pagine.

Nella gestione delle strutture dati che cambiano di dimensione è di aiuto la segmentazione, la quale semplifica il collegamento e la condivisione; inoltre facilita l'uso di diversi meccanismi di protezione per diversi segmenti. Alcune volte la segmentazione e la paginazione sono combinate per fornire una memoria virtuale a due dimensioni: il sistema MULTICS e l'Intel Pentium supportano la segmentazione e la paginazione.

PROBLEMI

1. Un calcolatore ha abbastanza memoria per contenere quattro programmi nella sua memoria centrale. Questi programmi restano inattivi in attesa del completamento di operazioni di ingresso/uscita per metà del loro tempo. Che percentuale viene sprecata del tempo di CPU?
2. In Figura 4.21 abbiamo visto un esempio di come diversi job possano girare in parallelo e terminare più velocemente che se fossero fatti girare sequenzialmente. Si supponga che due job, ciascuno dei quali richiede 10 minuti di tempo di CPU, inizino simultaneamente. Quanto tempo impiegherà l'ultimo job per terminare se fossero fatti girare sequenzialmente? Quanto tempo se fossero fatti girare in parallelo? Si assuma il 50% di tempo di attesa per operazioni di ingresso/uscita.
3. Alcuni sistemi di swapping tentano di eliminare la frammentazione interna tramite la compattazione. Supponiamo di avere una distribuzione casuale di diversi buchi e di diversi segmenti di dati e di avere 10 ns di tempo per leggere e scrivere una parola di memoria da 32 bit, quanto tempo si dovrebbe impiegare per compattare 128 MB di memoria? Per semplicità, si assuma che la parola 0 sia parte di un buco e che la parola più alta in memoria contenga dati validi.
4. In questo problema si deve confrontare lo spazio richiesto per tenere traccia della memoria libera utilizzando una mappa di bit ed utilizzando una lista linkata. La memoria di 128 MB è allocata in unità di n byte. Per la lista linkata, si supponga che la memoria sia fatta da una sequenza alternata di segmenti e buchi, ciascuno di 64 KB. Inoltre si supponga che ogni nodo nella lista linkata necessiti di un indirizzo di memoria di 32 bit, una lunghezza di 16 bit, ed un campo per il nodo successivo di 16 bit. Quanti byte di memoria sono richiesti per ogni metodo? Quale dei due è migliore?
5. Si consideri un sistema con swapping nel quale la memoria si compone di buchi della seguente dimensione (in ordine di indirizzo): 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB e 15KB. Quale buco viene preso per richieste successive di segmenti di:
 - (a) 12KB
 - (b) 10KB
 - (c) 9KB
 con l'algoritmo first fit? Poi ripetete l'esercizio nel caso del best fit, del worst fit e del next fit.
6. Che differenza c'è tra un indirizzo fisico ed un indirizzo virtuale?

7. Per ognuno dei seguenti indirizzi virtuali decimali, si calcoli il numero di pagina virtuale e l'offset per una pagina di 4KB e per una pagina di 8KB: 20000, 32768, 60000.
8. Usando la tabella delle pagine della Figura 4.10, dare l'indirizzo fisico corrispondente a ciascuno dei seguenti indirizzi virtuali:
 - (a) 20
 - (b) 4100
 - (c) 8300
9. Il processore Intel 8086 non supporta la memoria virtuale. Ciò nonostante, alcune ditte hanno commercializzato sistemi che contenevano un 8086 non modificato e che attuavano la paginazione. Fare una supposizione sensata su come ci sono riusciti. (Suggerimento: si pensi alla locazione logica della MMU.)
10. L'ammontare di spazio su disco che deve essere disponibile per la memorizzazione delle pagine è relativo al massimo numero di processi, n , al numero di byte per lo spazio di indirizzamento virtuale, v , ed al numero di byte di RAM, r . Dare una espressione dello spazio su disco richiesto nel caso peggiore. Quanto è realistico questo ammontare di spazio?
11. Se una istruzione impiega 10 ns, ed un fault di pagina impiega altri n ns, dare una formula che esprima il tempo effettivo di una istruzione nell'ipotesi che avvenga un fault di pagina ogni k istruzioni.
12. Una macchina ha uno spazio di indirizzamento di 32 bit e una dimensione delle pagine di 8KB. La tabella delle pagine è interamente gestita dall'hardware, con una parola da 32 bit per ognuno degli elementi. Quando si fa partire un processo, la tabella delle pagine viene copiata dalla memoria all'hardware ad una velocità di una parola ogni 100 ns. Se ciascun processo gira per 100 ms (compreso il tempo necessario per il caricamento della tabella delle pagine), qual è la frazione del tempo di CPU dedicata al caricamento della tabella delle pagine?
13. Un calcolatore con indirizzi da 32 bit usa una tabella delle pagine a due livelli. L'indirizzo virtuale viene diviso in un campo da 9 bit per la tabella delle pagine di primo livello, in un campo da 11 bit per la tabella delle pagine di secondo livello e in un campo di offset. Quanto sono grandi le pagine e quante ve ne sono all'interno dello spazio di indirizzamento virtuale?
14. Supponiamo che un indirizzo virtuale da 32 bit venga spezzato in quattro campi, a , b , c e d . I primi tre vengono usati per un sistema di paginazione con tabelle delle pagine a tre livelli. Il campo d è l'offset. Il numero delle pagine dipende dall'ampiezza di tutti e quattro i campi? Se no, da quali dipende e da quali no?
15. Un calcolatore ha uno spazio di indirizzamento virtuale di 32 bit e pagine da 4KB. Il programma e i dati sono inseriti insieme nella pagina più bassa (0-4095). Lo stack è inserito nella pagina più alta. Quanti elementi sono necessari nella tabella delle pagine se viene utilizzata una paginazione tradizionale (ad un livello)? Quanti elementi della tabella delle pagine sono necessari per una paginazione a due livelli, con 10 bit in ogni parte?
16. Di seguito viene riportata una breve traccia di esecuzione di un frammento di programma per un calcolatore con pagine da 512 byte. Il programma si trova all'indirizzo 1020, il suo stack si trova all'indirizzo 8192 (e cresce verso lo 0). Si dia la stringa dei riferimenti di pagina generata da questo programma. Ciascuna istruzione

occupa 4 byte (1 parola) comprese le costanti immediate, e sia le istruzioni sia i riferimenti ai dati vanno elencati nella stringa dei riferimenti di pagina.

Carica la parola all'indirizzo 6144 nel registro 0

Metti il registro 0 sulla pila

Chiama la procedura all'indirizzo 5120, mettendo sulla pila l'indirizzo di ritorno

Sottrai la costante immediata 16 dal puntatore della pila

Confronta il parametro attuale alla costante immediata 4

Se uguale, salta all'indirizzo 5152

17. Un calcolatore i cui processi hanno nel loro spazio di indirizzamento 1024 pagine, mantiene in memoria le sue tabelle delle pagine. L'overhead richiesto per leggere una parola dalla tabella delle pagine è di 5 ns. Per ridurre questo overhead, il calcolatore dispone di una memoria associativa che contiene 32 coppie (pagina virtuale, numero di pagina fisica) e che può eseguire una ricerca in 1 ns. Qual è l'hit rate necessario per ridurre l'overhead medio a 2 ns?
18. Il TLB del VAX non contiene il bit *R*. Perché?
19. Come può essere implementato tramite hardware il dispositivo della memoria associativa per un TLB, e quali sono le implicazioni in termini di espandibilità che questa realizzazione comporterebbe?
20. Una macchina ha indirizzi virtuali di 48 bit e indirizzi fisici di 32 bit, e le pagine sono da 8 KB. Quanti elementi sono necessari per la tabella delle pagine?
21. Un calcolatore con pagine da 8 KB, una memoria principale da 256 KB, ed uno spazio di indirizzamento virtuale da 64 GB utilizza una tabella delle pagine invertite per implementare la sua memoria virtuale. Quanto dovrebbe essere grande la tabella hash per assicurare una lunghezza media della catena degli elementi inferiore ad 1? Si supponga che la dimensione della tabella hash sia una potenza di due.
22. Uno studente di un corso di progettazione di compilatori propone al professore il progetto di un compilatore che produceva una lista di riferimenti alle pagine che poteva essere utilizzata per implementare l'algoritmo ottimale di rimpiazzamento delle pagine. Tutto questo è possibile? Perché sì o perché no? Si potrebbe fare qualcosa per migliorare l'efficienza di paginazione a tempo di esecuzione?
23. Se si utilizza un algoritmo di rimpiazzamento FIFO con quattro pagine fisiche e otto pagine virtuali, quanti fault di pagina si avranno con la stringa dei riferimenti 0172327103 se le quattro pagine fisiche sono inizialmente vuote? Ripetere l'esercizio per l'algoritmo LRU.
24. Si consideri la sequenza di pagine di Figura 4.16(b). Si supponga che i bit *R* dalla pagina *B* alla pagina *A* siano rispettivamente 11011011. Quale pagina rimuoverà l'algoritmo della seconda opportunità?
25. Un piccolo calcolatore ha quattro pagine fisiche. Al primo ciclo di clock, i bit *R* sono 0111 (per la pagina 0 vale 0, per le altre vale 1). Ai cicli di clock successivi i valori sono 1011, 1010, 1101, 0010, 1010, 1100 e 0001. Nell'ipotesi che venga usato l'algoritmo dell'invecchiamento con un contatore a 8 bit, date i valori dei quattro contatori dopo l'ultimo ciclo di clock.
26. Supponiamo di avere $\tau = 400$ per la Figura 4.21. Quale pagina sarà rimossa?

27. Nell'algoritmo del WSClock di Figura 4.22(c), la lancetta punta alla pagina con il bit $R = 0$. Se $\tau = 400$, la pagina verrà rimossa? Cosa accadrebbe con $\tau = 1000$?
28. Quanto tempo si impiega per caricare un programma da 64KB da un disco con un tempo medio di seek di 10 ms e un tempo di rotazione di 10 ms, le cui tracce hanno una capacità di 32 KB
- per una dimensione delle pagine di 2 KB?
 - per una dimensione delle pagine di 4 KB?

Le pagine sono sparpagliate sul disco in maniera casuale e il numero dei cilindri è talmente grande che la probabilità che due pagine stiano sullo stesso cilindro è trascurabile.

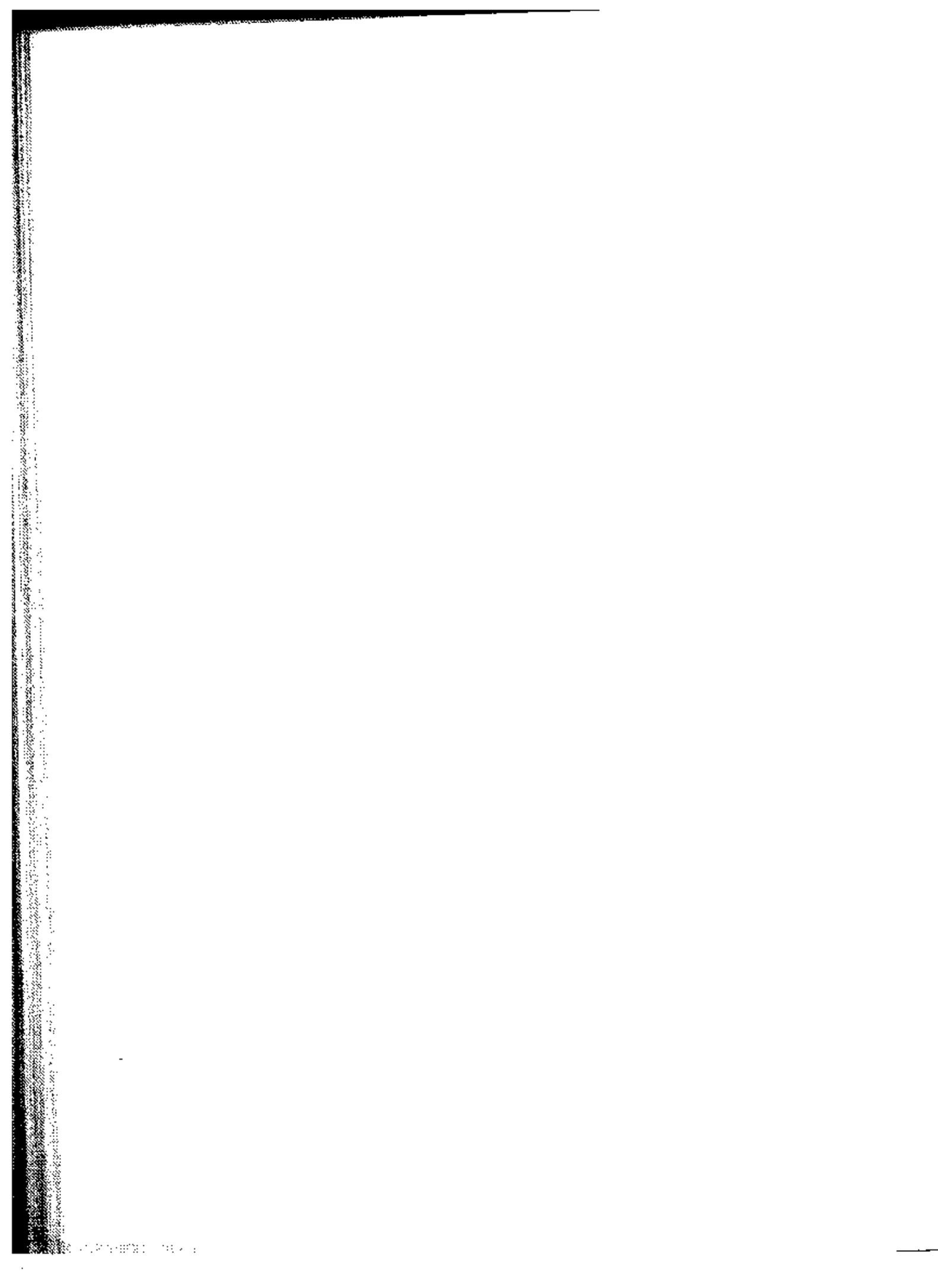
29. Un calcolatore ha quattro pagine fisiche. Il tempo del caricamento, dell'ultimo accesso e i bit R e M per ciascuna delle pagine sono mostrati qui sotto (i tempi sono in cicli di clock).

Pagina	Caricamento	Ultimo riferimento	R	M
0	126	280	1	0
1	230	265	0	0 - 1
2	140	270	0	0
3	110	285	1	1

- Qual è la pagina rimpiazzata con NRU?
 - Qual è la pagina rimpiazzata con FIFO?
 - Qual è la pagina rimpiazzata con LRU?
 - Qual è la pagina rimpiazzata con l'algoritmo della seconda opportunità?
30. Una delle prime macchine a condivisione di tempo, il PDP-1, aveva una memoria di 4K parole da 18 bit. Ad ogni istante manteneva un solo processo in memoria, e quando lo scheduler decideva di mandare in esecuzione un altro processo, il processo in memoria veniva scritto su un cilindro. Il cilindro può cominciare a scrivere (o a leggere) ad indirizzi arbitrari, piuttosto che solo all'indirizzo 0. Perché secondo voi è stato scelto questo cilindro?
31. Un calcolatore mette a disposizione di ciascun processo 65536 byte di spazio di indirizzamento divisi in pagine da 4096 byte. Un certo programma ha un testo di 32768 byte e una dimensione dei dati di 16384 byte e una dimensione della pila di 15870 byte. Questo programma può girare nello spazio di indirizzamento? Se la dimensione delle pagine fosse di 512 pagine, potrebbe girare? Si ricordi che una pagina non può contenere parti di due segmenti diversi.
32. Una pagina può essere contemporaneamente in due working set? Spiega.
33. Se una pagina è condivisa tra due processi, è possibile che la pagina sia solo in lettura (read-only) per un processo e in lettura e scrittura (read-write) per l'altro? Perché sì o perché no?
34. È stato osservato che il numero delle istruzioni eseguite fra due fault di pagina è direttamente proporzionale al numero delle pagine fisiche allocate per il programma. Se si raddoppia la memoria disponibile, l'intervallo fra due fault viene anch'esso raddoppiato. Supponiamo che una istruzione normale impieghi 1 microsecondo, ma che se avviene un fault di pagina ne impieghi 2001 (cioè 2 ms per gestire il fault). Se un programma impiega 60 secondi per girare, durante i quali genera 15000 fault di pagina, per quanto tempo girerebbe se la memoria venisse raddoppiata?

35. Un gruppo di progettisti di sistemi operativi della Compagnia dei Calcolatori Frugali sta pensando a come ridurre la quantità di memoria di backup necessaria nel loro nuovo sistema operativo. Il capo guru ha appena suggerito di non preoccuparsi di salvare il testo del programma nell'area di swap, ma di paginarlo direttamente dal file binario ogni volta che risultasse necessario. Sotto quali condizioni, se esistono, quest'idea funziona per il testo del programma? Sotto quali condizioni, se esistono, funziona per i dati?
36. Una istruzione in linguaggio macchina per caricare una parola di 32 bit in un registro contiene l'indirizzo di 32 bit della parola da caricare. Qual è il numero massimo di fault di pagina che questa istruzione può causare?
37. Spiegare la differenza fra la frammentazione interna e la frammentazione esterna. Quale delle due è presente nei sistemi a paginazione? E quale delle due si verifica nei sistemi con segmentazione pura?
38. Quando si usa sia la paginazione sia la segmentazione, come accade in MULTICS, occorre prima cercare il segmento dei descrittori e poi quello della pagina. Il TLB funziona anche in questo caso, con due livelli di ricerca?
39. Disegna un istogramma e calcola la media e la mediana delle dimensioni dei file binari eseguibili su un computer a cui hai accesso. Su un sistema Windows, osserva tutti i file .exe e .dll; su un sistema UNIX osserva tutti i file eseguibili in /bin, /usr/bin, e /local/bin che non siano script (oppure utilizza l'utility *file* per trovare tutti i file eseguibili). Determina la dimensione di pagina ottimale per questo calcolatore considerando il codice (non i dati). Considera la frammentazione interna e la dimensione dalla tabella delle pagine, facendo alcune ipotesi ragionevoli sulla dimensione di un elemento della tabella delle pagine. Supponi che tutti i programmi abbiano la stessa probabilità di essere eseguiti e perciò abbiano lo stesso peso.
40. Piccoli programmi per MS-DOS possono essere compilati come file .COM, questi file sono sempre caricati all'indirizzo 0x100 in un solo segmento di memoria che è utilizzato per il codice, i dati e lo stack. Le istruzioni che trasferiscono il controllo dell'esecuzione, come ad esempio JMP e CALL, o quelle che accedono ai dati statici da indirizzi fissi, hanno gli indirizzi compilati nel codice oggetto. Scrivi un programma che può rilocare tale file di programma in modo da farlo girare partendo da un indirizzo arbitrario. Il programma deve scandire il codice cercando i codici oggetto per le istruzioni che riferiscono indirizzi fissi di memoria, quindi deve modificare gli indirizzi che puntano alle locazioni di memoria all'interno dell'intervallo in cui devono essere rilocati. Puoi trovare i codici oggetto in un testo di programmazione in linguaggio assembler. Nota che, in generale, effettuare perfettamente tutto il procedimento, senza utilizzare nessuna informazione aggiuntiva è un compito impossibile, poiché alcune parole di dato possono avere lo stesso valore dei codici oggetto delle istruzioni.
41. Scrivi un programma che simula un sistema di paginazione. All'inizio del programma, si dovrebbe chiedere di scegliere un algoritmo di rimpiazzamento delle pagine, scegliendo tra FIFO, LRU, e un altro a scelta. Ad ogni ciclo, leggi il numero delle pagine riferite da un file. Genera una lista simile a quella di Figura 4.25, ma ruotata di 90 gradi, in modo che ogni nuovo riferimento ad una pagina aggiunga una riga al file di output.
42. Scrivi un programma che modelli l'algoritmo della stringa delle distanze descritto nel testo. L'input del programma è una lista di riferimenti alle pagine (conte-

nuta in un file), più il numero di pagine fisiche disponibili nella memoria fisica. Se possibile, utilizza una traccia dei dati dai programmi reali invece di utilizzare riferimenti a pagine generati in maniera casuale. Il programma dovrebbe mantenere lo stack delle pagine, analogamente alla Figura 4.25. Ad ogni fault di pagina, dovrebbe essere chiamata una procedura per scegliere un rimpiazzamento delle pagine. Quando l'esecuzione è completata, il programma deve disegnare la stringa delle distanze, analogamente alla Figura 4.26. Esegui diverse esecuzioni per differenti valori della dimensione della memoria, e guarda quali conclusioni puoi trarre.



INPUT/OUTPUT

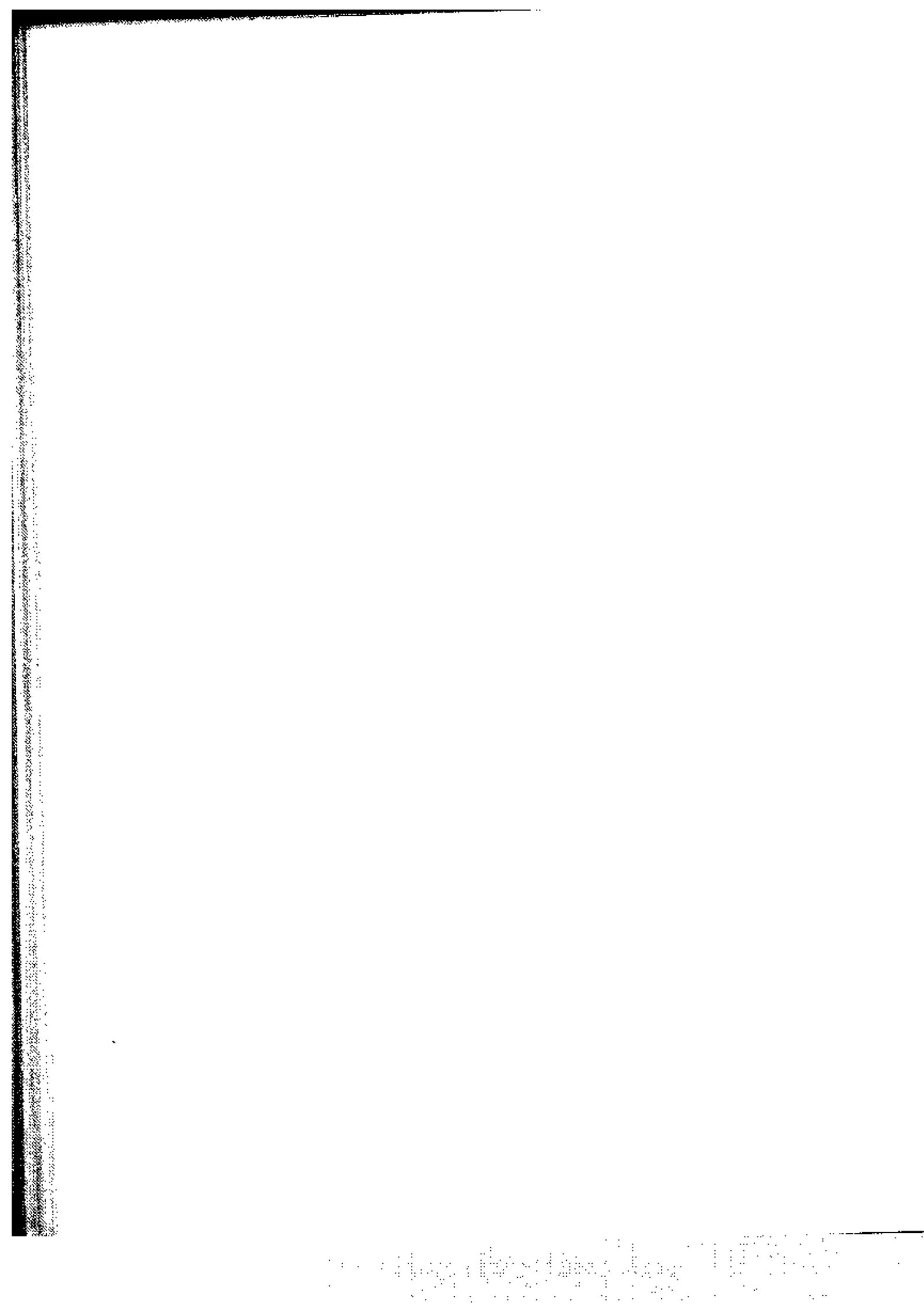
Una delle funzioni principali di un sistema operativo è il controllo dei dispositivi di input/output (ingresso/uscita): esso deve inviare i comandi ai dispositivi, catturare le interruzioni e trattare le condizioni di errore. Deve anche fornire un'interfaccia fra i dispositivi fisici e il resto del sistema che sia semplice e facile da usare; per quanto possibile, l'interfaccia dovrà essere la stessa per tutti i dispositivi (indipendenza dal dispositivo). Il codice relativo all'I/O rappresenta una porzione significativa dell'intero sistema operativo; questo capitolo parla del modo in cui il sistema operativo gestisce l'I/O.

Il contenuto del capitolo sarà il seguente: prima si esamineranno brevemente i principi generali dei dispositivi fisici di I/O, e poi, in generale, il software di I/O. Esso può essere strutturato a livelli, in cui ogni livello ha un ben preciso compito da svolgere. Si guarderanno questi livelli per capire cosa fanno e come collaborano.

Dopo tale introduzione, si esamineranno in dettaglio diversi dispositivi: i dischi, i clock, le tastiere e i monitor video, dettagliando per ciascuno di essi sia l'hardware sia il software. Infine, si considererà la gestione del consumo di energia.

5.1 Principi dell'hardware di I/O

Personne diverse considerano l'hardware relativo all'I/O da punti di vista differenti. Gli ingegneri elettronici lo vedono in termini di circuiti integrati, fili, alimentatori, motori ed altri componenti fisici che costituiscono l'hardware stesso. I programmati vedono l'interfaccia presentata al software - i comandi che l'hardware accetta, le funzioni che è in-



INPUT/OUTPUT

Una delle funzioni principali di un sistema operativo è il controllo dei dispositivi di input/output (ingresso/uscita): esso deve inviare i comandi ai dispositivi, catturare le interruzioni e trattare le condizioni di errore. Deve anche fornire un'interfaccia fra i dispositivi fisici e il resto del sistema che sia semplice e facile da usare; per quanto possibile, l'interfaccia dovrà essere la stessa per tutti i dispositivi (indipendenza dal dispositivo). Il codice relativo all'I/O rappresenta una porzione significativa dell'intero sistema operativo; questo capitolo parla del modo in cui il sistema operativo gestisce l'I/O.

Il contenuto del capitolo sarà il seguente: prima si esamineranno brevemente i principi generali dei dispositivi fisici di I/O, e poi, in generale, il software di I/O. Esso può essere strutturato a livelli, in cui ogni livello ha un ben preciso compito da svolgere. Si guarderanno questi livelli per capire cosa fanno e come collaborano.

Dopo tale introduzione, si esamineranno in dettaglio diversi dispositivi: i dischi, i clock, le tastiere e i monitor video, dettagliando per ciascuno di essi sia l'hardware sia il software. Infine, si considererà la gestione del consumo di energia.

5.1 Principi dell'hardware di I/O

Persone diverse considerano l'hardware relativo all'I/O da punti di vista differenti. Gli ingegneri elettronici lo vedono in termini di circuiti integrati, fili, alimentatori, motori ed altri componenti fisici che costituiscono l'hardware stesso. I programmati vedono l'interfaccia presentata al software - i comandi che l'hardware accetta, le funzioni che è in

grado di fornire, e gli errori che possono essere restituiti. In questo libro verrà trattata la programmazione dei dispositivi di I/O e non la loro progettazione, né la costruzione o la manutenzione, sicché il nostro interesse si concentrerà su come l'hardware è programmato e non su come lavora internamente. Tuttavia, la programmazione di molti dispositivi di I/O è spesso intimamente legata al loro funzionamento interno: nelle prossime tre sezioni si daranno alcune nozioni di base relative all'hardware per l'I/O in connessione con la programmazione; questa parte può essere considerata come un ripasso e un'estensione del materiale introduttivo della Sezione 1.4.

5.1.1 Dispositivi di I/O

I dispositivi di I/O possono essere divisi genericamente in due categorie: i **dispositivi a blocchi** e i **dispositivi a caratteri**. Un dispositivo a blocchi memorizza le informazioni in blocchi di lunghezza fissa, ognuno dei quali ha un proprio indirizzo; normalmente la lunghezza dei blocchi va da 512 a 32768 byte. La proprietà essenziale di un

Dispositivo	Velocità dati
Tastiera	10 byte/s
Mouse	100 byte/s
Modem a 56 K	7 KB/s
Canale telefonico	8 KB/s
Linee duali ISDN	16 KB/s
Stampante laser	100 KB/s
Scanner	400 KB/s
Ethernet classica	1,25 MB/s
USB (Universal Serial Bus)	1,5 MB/s
Telecamera digitale	4 MB/s
Disco IDE	5 MB/s
CD-ROM 40x	6 MB/s
Fast Ethernet	12,5 MB/s
Bus ISA	16,7 MB/s
Disco FIDE (ATA-2)	16,7 MB/s
FireWire (IEEE 1394)	50 MB/s
Monitor XGA	60 MB/s
Rete SONET OC-12	78 MB/s
Disco SCSI Ultra-2	80 MB/s
Ethernet Gigabit	125 MB/s
Nastro Ultrium	320 MB/s
Bus PCI	528 MB/s
Sun Gigaplane XB backplane	20 GB/s

Figura 5.1 Esempi di velocità di dati tipiche di alcuni dispositivi, reti e bus.

dispositivo a blocchi è che è possibile leggere o scrivere ciascun blocco in maniera indipendente dagli altri; i più comuni dispositivi a blocchi sono i dischi.

A ben vedere, il confine tra i dispositivi che sono indirizzabili a blocchi e quelli che non lo sono, non è ben definito. Tutti concordano sul fatto che un disco sia un dispositivo indirizzabile a blocchi perché, indipendentemente da dove si trovi il braccio della testina, è sempre possibile spostarsi su un altro cilindro e poi aspettare che il blocco richiesto passi sotto la testina. Ora si consideri un nastro magnetico usato per fare backup del contenuto del disco: i nastri contengono sequenze di blocchi e se si dà al controllore del nastro il comando di lettura del blocco N , esso può comunque riavvolgere il nastro e andare poi avanti fino a raggiungere il blocco N . Quest'operazione è analoga a quella che il disco fa durante lo spostamento del braccio della testina, tranne per il fatto che richiede più tempo. Poi, non è detto che sia possibile riscrivere un blocco che si trova a metà nastro, per cui, anche se fosse possibile usare i nastri come dispositivi a blocchi, sarebbe una certa forzatura: questi dispositivi non sono normalmente usati in questa maniera.

L'altro tipo di dispositivo di I/O è il dispositivo a caratteri: un dispositivo di questo genere invia o riceve un flusso di caratteri senza tener conto di alcuna struttura di blocco, non è indirizzabile, e non può eseguire alcuna operazione di posizionamento. Le stampanti, le interfacce di rete, i mouse (per puntare), i topi (per gli esperimenti dei laboratori di psicologia) e la maggior parte degli altri dispositivi che non assomigliano ai dischi, possono essere visti come dispositivi a carattere.

Questo schema di classificazione non è perfetto, infatti alcuni dispositivi non vi rientrano. I clock, per esempio, non sono indirizzabili a blocchi, ma non generano né ricevono flussi di caratteri; essi non fanno altro che causare interruzioni a intervalli di tempo ben definiti. Neppure i dispositivi video a mappa di memoria appartengono ai modelli dei dispositivi a blocchi; tuttavia, il modello dei dispositivi a blocchi o a caratteri è abbastanza generale da poter essere usato come base per rendere la parte del sistema operativo che tratta l'I/O indipendente dal dispositivo. Il file system, per esempio, tratta con dispositivi a blocchi astratti, e lascia le parti dipendenti dal dispositivo al software di livello più basso.

I dispositivi di I/O coprono un'enorme gamma di velocità, e questo richiede al software di dover dare buone prestazioni su frequenze di dati di diversi ordini di grandezza. La Figura 5.1 mostra frequenze di dati per alcuni dispositivi comuni; molti di essi tendono a diventare sempre più veloci al passare del tempo.

5.1.2 I controllori dei dispositivi

Le unità di I/O sono costituite, tipicamente, da una componente meccanica ed una elettronica ed è spesso possibile separare le due parti per avere un progetto più generale e modulare. La componente elettronica è chiamata **controllore del dispositivo** o **adattatore**. Sui personal computer, essa prende spesso la forma di una scheda a circuito stampato che va inserita all'interno del calcolatore all'interno di uno slot di espansione; la componente meccanica è il dispositivo stesso. Questa struttura è mostrata in Figura 1.5.

La scheda del controllore è fornita di solito di un connettore, nel quale può essere inserito il cavo proveniente dal dispositivo stesso; molti controllori possono gestire due, quattro o anche otto dispositivi identici. Se l'interfaccia tra il controllore e il dispositivo è standard, o secondo uno standard ufficiale, come ANSI, IEEE o ISO, o secondo uno standard de facto, allora i produttori possono fabbricare controllori o dispositivi conformi a quell'interfaccia. Per esempio, molti produttori costruiscono dispositivi a disco che possono funzionare con l'interfaccia IDE o SCSI.

L'interfaccia fra il controllore e il dispositivo è spesso a bassissimo livello; un disco, per esempio, può essere formattato con 256 settori di 512 byte per traccia. Ciò che esce realmente dal dispositivo fisico, è un flusso seriale di bit che comincia con un **preamble** o sincronizzatore iniziale seguito dai 4096 bit di un settore e infine un **checksum** o **codice correttore di errore** (error correction code, ECC). Il preamble è scritto quando il disco viene formattato e contiene il numero di cilindro e di settore, la lunghezza del settore ed altri dati simili, oltre ad informazioni di sincronizzazione.

Il compito del controllore è di convertire il flusso seriale di bit in un blocco di byte, e di eseguire la correzione degli errori, se necessaria. Prima il blocco dei byte è ricostruito un bit alla volta in un buffer che si trova all'interno dello stesso controllore e, dopo che il suo checksum è stato controllato e che il blocco è stato dichiarato privo di errori, esso può essere copiato all'interno della memoria principale.

Il controllore di un terminale video CRT lavora come un dispositivo seriale a bit, ad un livello altrettanto basso. Esso legge dalla memoria i byte che contengono i caratteri da stampare sullo schermo, e genera i segnali necessari a modulare il pennello del tubo catodico, affinché esso scriva sullo schermo. Il controllore genera anche i segnali necessari affinché il pennello elettronico del tubo catodico ritorni indietro orizzontalmente dopo la scansione di ogni singola linea, e i segnali per il ritorno verticale dopo la scansione di ogni singolo quadro. Se non ci fosse il controllore del terminale video, il programmatore del sistema operativo dovrebbe esplicitamente programmare la scansione del tubo catodico; in presenza del controllore, il sistema operativo inizializza il controllore con pochi parametri, come il numero di caratteri o pixel per linea e il numero di linee per ogni schermo, poi lascia che sia il controllore a prendersi cura di guidare il pennello elettronico.

5.1.3 I/O mappato in memoria

Ogni controllore possiede alcuni registri che sono usati per comunicare con la CPU. Scrivendo in questi registri, il sistema operativo può chiedere al dispositivo di spedire o accettare dati, di accendersi o spegnersi, oppure di eseguire qualche azione. Leggendo il contenuto di questi registri, il sistema operativo può conoscere lo stato del dispositivo, capire se esso è pronto a ricevere un comando e così via.

Oltre ai registri di controllo, molti dispositivi sono dotati di un buffer dati, che può essere letto o scritto dal sistema operativo; ad esempio, comunemente i computer visualizzano i pixel sullo schermo utilizzando una RAM video, che in pratica è un buffer dati dentro il quale programmi e sistema operativo possono scrivere.

Se adesso ci chiediamo come la CPU possa comunicare con i registri del controllore e con i buffer dati del dispositivo, scopriamo due alternative. Nel primo approccio, ogni registro del controllore è caratterizzato da un numero di **porta I/O** (un intero a 8 o 16 bit). Usando una speciale istruzione di I/O del tipo

IN REG, PORTA

la CPU può leggere il registro del controllore PORTA, ed immagazzinare il risultato nel registro di CPU REG. In modo simile, usando il comando

OUT PORTA, REG

la CPU può scrivere il contenuto di REG in un registro del controllore. La maggior parte dei calcolatori della vecchia generazione, tra cui la maggior parte dei mainframe quali l'IBM 360 e i suoi successori, lavoravano in questo modo.

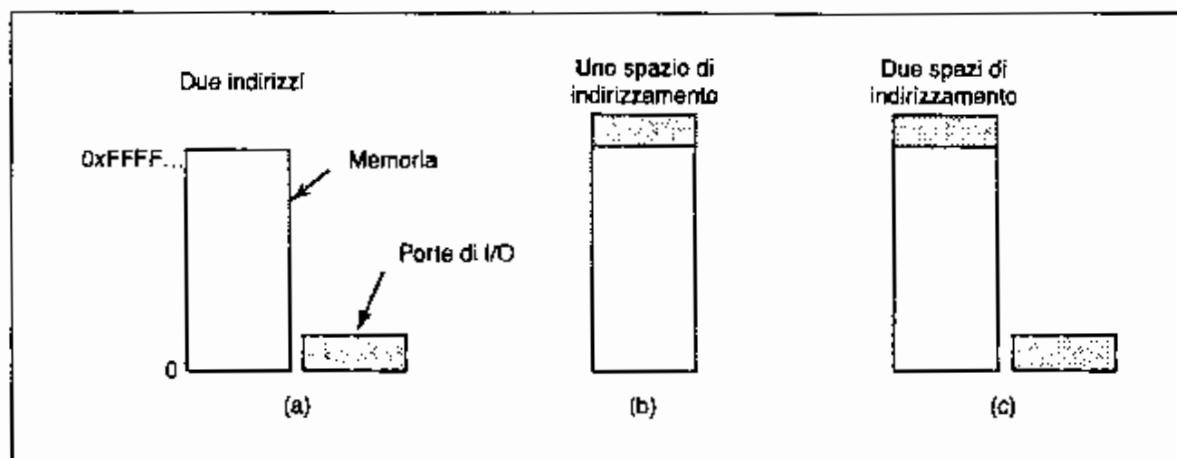


Figura 5.2 (a) Spazio di memoria e di I/O separati. (b) I/O mappato in memoria. (c) Irido.

In questo schema, gli spazi di indirizzamento per l'I/O e la memoria sono diversi, come mostrato in Figura 5.2(a). In questo tipo di progettazione, le istruzioni

IN R0, 4

e

MOV R0, 4

sono completamente diverse: la prima legge il contenuto della porta di I/O 4 e lo mette in R0, mentre la seconda legge il contenuto della parola in memoria 4 e lo mette in R0. I 4 che compaiono in questi esempi si riferiscono quindi a spazi di memoria diversi e che non hanno nulla a che fare l'uno con l'altro.

Il secondo approccio, introdotto con il PDP-11, mappa i registri del controllore nello spazio di indirizzamento della memoria centrale, come mostrato in Figura 5.2(b); ad ogni registro del controllore viene assegnato un indirizzo di memoria unico, a cui non corrisponde nessuna parola di memoria. Questo sistema è chiamato **I/O mappato in memoria**. Solitamente gli indirizzi assegnati all'I/O sono in cima allo spazio di indirizzamento. Uno schema ibrido, con l'I/O del buffer dati mappato in memoria, e porte I/O separate per i registri del controllore, è mostrato in Figura 5.2(c): il Pentium usa questa architettura, con porte I/O dallo 0 a 64K e, in aggiunta, indirizzi da 640K a 1M riservati ai buffer dati dei dispositivi per calcolatori IBM compatibili.

Come funziona questo schema? In tutti i casi, quando la CPU vuole leggere una parola, sia da memoria sia da una porta di I/O, mette l'indirizzo di cui ha bisogno sulle linee di indirizzamento dei bus e poi imposta un segnale READ sulla linea di controllo del bus. Una seconda linea è utilizzata per specificare se la richiesta riguarda spazio di I/O o spazio di memoria: se si tratta di spazio di memoria, allora la memoria risponderà alla richiesta, se si tratta di spazio di I/O, il dispositivo di I/O risponderà alla richiesta. Se, come in Figura 5.2(b), si ha solo spazio di memoria, allora ogni modulo di memoria e ogni dispositivo di I/O confronta le linee di indirizzamento con il gruppo di indirizzi in suo possesso e, se l'indirizzo in questione ricade in tale gruppo, risponde alla richiesta. Siccome non è possibile che un indirizzo venga assegnato sia alla memoria sia a un dispositivo di I/O, non si verificano ambiguità e conflitti.

I due schemi per effettuare l'indirizzamento ai controllori hanno diversi lati forti e

diversi lati deboli. Iniziamo con l'analizzare i vantaggi dell'I/O mappato in memoria: primo, se sono necessarie delle istruzioni speciali di I/O per leggere e scrivere i registri del controllore, occorre un qualche tipo di linguaggio macchina (assembler) in quanto non c'è alcun modo di eseguire un IN o un OUT con un'istruzione in C o C++; e la chiamata di queste procedure aggiunge un overhead al controllo dell'I/O. Invece, con l'I/O mappato in memoria, i registri dei controllori sono semplicemente delle variabili in memoria, che possono venire indirizzate in C esattamente come qualsiasi altra variabile, quindi con un I/O mappato in memoria un controllore di dispositivo può essere scritto interamente in C; viceversa, senza I/O mappato in memoria, bisogna utilizzare del codice assembler.

Secondo, con l'I/O mappato in memoria, non occorrono meccanismi di protezione speciali per impedire ai processi utente di eseguire dell'I/O; quello che il sistema operativo deve fare è evitare di mettere la parte di spazio di indirizzamento contenente i registri, in uno spazio di indirizzamento virtuale utente. Ancora meglio, se i registri di diversi controllori si trovano su pagine diverse dello spazio di indirizzamento, il sistema operativo può lasciare all'utente il controllo di certi dispositivi ma non di altri, semplicemente includendo le pagine necessarie nella tabella delle pagine dell'utente. Questo schema permette di mettere diversi controllori di dispositivo in diversi spazi di indirizzamento, non solo riducendo la dimensione del kernel, ma anche impedendo ad un controllore di interferire con altri.

Terzo, con l'I/O mappato in memoria ogni istruzione che può fare riferimento in memoria può anche fare riferimento ai registri dei controllori; ad esempio, un'istruzione TEST che controlla se una parola in memoria è 0, può anche venire usata per controllare se un registro di controllore è 0, cosa che può accadere quando il dispositivo è inattivo e può quindi accettare un nuovo comando. Il codice, scritto in linguaggio macchina, potrebbe avere il seguente aspetto:

```
LOOP:    TEST PORTA_4      // controlla che la porta 4 sia 0
          BEQ READY        // se è 0, vai a READY
          BRANCH LOOP       // altrimenti, continua il test
READY:
```

In assenza di I/O mappato in memoria, il registro del controllore deve essere prima copiato nella CPU e poi testato, il che richiede due istruzioni invece di una. Nel caso del codice scritto sopra, occorre aggiungere al loop una quarta istruzione, rallentando il processo di ricerca di un dispositivo inattivo.

Come spesso accade nella progettazione di calcolatori, anche qui occorre fare dei compromessi. L'I/O mappato in memoria ha anche i suoi svantaggi: primo, la maggior parte dei calcolatori odierni hanno una qualche forma di cache di memoria; ma sarebbe disastroso mettere nella cache un registro di dispositivo. Consideriamo il loop in codice macchina descritto in precedenza, in presenza di caching: al primo riferimento a PORTA_4 esso verrebbe messo nella cache, e i successivi riferimenti non farebbero altro che accedere alla cache e non interpellerebbero più il dispositivo; quando il dispositivo diventa inattivo (READY), il software non ha alcun modo di accorgersene e il ciclo prosegue all'infinito.

Per impedire questa situazione, l'hardware deve essere dotato di una funzionalità per la disabilitazione selettiva delle capacità di usare la cache, ad esempio pagina per pagina. Questa caratteristica aggiunge complessità sia all'hardware sia al sistema operativo, il quale deve gestire il caching selettivo.

Secondo, se c'è un solo spazio di indirizzamento, allora tutti i moduli della memoria e tutti i dispositivi di I/O devono esaminare tutti i riferimenti alla memoria e decidere a

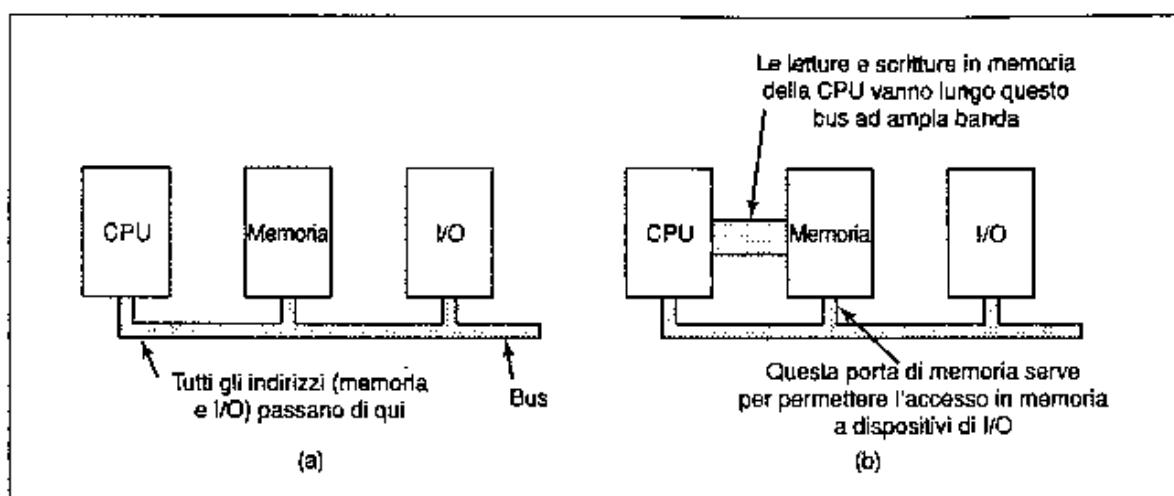


Figura 5.3 [a] Architettura a singolo bus. [b] Architettura a doppio bus, di cui un bus per la memoria.

quali rispondere; se il calcolatore è a singolo bus, come quello mostrato in Figura 5.3(a), allora è facile per ognuno analizzare ogni indirizzo.

Comunque, la tendenza dei computer moderni è quella di avere un bus dedicato per la memoria ad alta velocità, come descritto in Figura 5.3(b), una proprietà che tra l'altro caratterizza anche i mainframe. Questo bus è pensato per ottimizzare le prestazioni della memoria, e non accetta compromessi a causa di dispositivi di I/O lenti; i sistemi Pentium hanno tre bus esterni (memoria, PCI e ISA), come mostrato in Figura 1.11.

Avere bus di memoria separati su macchine mappate in memoria è un problema, in quanto i dispositivi di I/O non hanno modo di vedere gli indirizzi di memoria, che viaggiano sul bus di memoria, e quindi non hanno modo di rispondere. Anche in questo caso sono state prese delle contromisure allo scopo di far lavorare l'I/O mappato in memoria su un sistema con bus multipli: una possibilità è quella di mandare prima tutti i riferimenti in memoria alla memoria e, se questa non risponde, la CPU li manda agli altri bus. Questa idea può funzionare, ma complica ulteriormente l'hardware.

Una seconda idea è quella di aggiungere un dispositivo "spia" che controlli il bus di memoria e che mandi tutti gli indirizzi che passano ai dispositivi di I/O potenzialmente interessati. In questo caso il problema è che i dispositivi di I/O possono non essere in grado di elaborare le richieste alla stessa velocità della memoria.

Una terza idea possibile, usata nella configurazione del Pentium in Figura 1.11, è di filtrare gli indirizzi nel circuito del bridge PCI. Questo circuito integrato contiene registri di intervallo che sono precaricati all'avvio del sistema; per esempio, l'intervallo da 640K a 1M potrebbe essere marcato come un intervallo non-memoria. Gli indirizzi che cadono in un intervallo marcato non-memoria potrebbero essere inviati al bus PCI invece che a quello di memoria. Lo svantaggio di questo schema è la necessità di capire al momento dell'avvio del sistema, quali indirizzi di memoria non sono in realtà indirizzi di memoria. In pratica ogni scheda ha lati positivi e lati negativi, quindi i compromessi sono inevitabili.

5.1.4 Accesso diretto in memoria

Indipendentemente dal fatto che la CPU faccia o meno I/O mappato in memoria, essa deve comunque indirizzare controlleri di dispositivo e scambiare dati con essi. La CPU

può richiedere dati ad un controllore di I/O un byte alla volta, ma questo causa una perdita di tempo alla CPU, quindi spesso si usa un diverso schema, chiamato **accesso diretto in memoria o DMA** (Direct Memory Access). Il sistema operativo può usare il DMA solo se l'hardware è fornito di un controllore DMA, cosa che molti sistemi hanno. Alcune volte questo controllore è integrato nei controllori di disco o altri controllori, ma questa configurazione richiede un controllore DMA separato per ogni dispositivo. Più comunemente, sul sistema è disponibile un solo controllore DMA (ad esempio sulla scheda madre), in modo da regolare trasferimenti a più di un dispositivo, spesso in modo corrente.

Indipendentemente dalla sua posizione fisica, il controllore DMA ha accesso al bus di sistema indipendentemente dalla CPU, come mostrato in Figura 5.4. Esso contiene variati registri che possono essere letti e scritti dalla CPU, tra cui un registro di indirizzo di memoria, un registro contatore di byte, uno o più registri di controllo. I registri di controllo specificano la porta I/O da usare, la direzione del trasferimento (lettura dal dispositivo di I/O o scrittura sul dispositivo di I/O), l'unità di trasferimento (byte per volta o parole per volta) e il numero di byte da trasferire in un burst.

Per spiegare come lavora il DMA, si esamina innanzitutto come avvengono le letture dal disco quando il DMA non è presente. Innanzitutto, il controllore legge il blocco dal disco (uno o più settori) serialmente, bit per bit, finché l'intero blocco non si trovi all'interno di un suo buffer. Subito dopo esegue il calcolo del checksum per verificare che non vi sia stato alcun errore di lettura, nel qual caso il controllore genera un'interruzione. Quando il sistema operativo comincia a girare, esso può leggere il blocco dal buffer del controllore, un byte o una parola alla volta, eseguendo un ciclo in cui, durante ogni iterazione, legge un byte o una parola dal registro del controllore e li scrive nella memoria centrale.

Quando si usa il DMA, la procedura è differente. Prima di tutto la CPU programma il controllore del DMA impostando i suoi registri in modo da sapere cosa e dove deve trasferire (passo 1 della Figura 5.4); inoltre invia un comando al controllore del disco, chiedendogli di copiare i dati dal disco al buffer interno e di controllare il checksum. Quando i dati validi si trovano all'interno del buffer del disco, il DMA può incominciare.

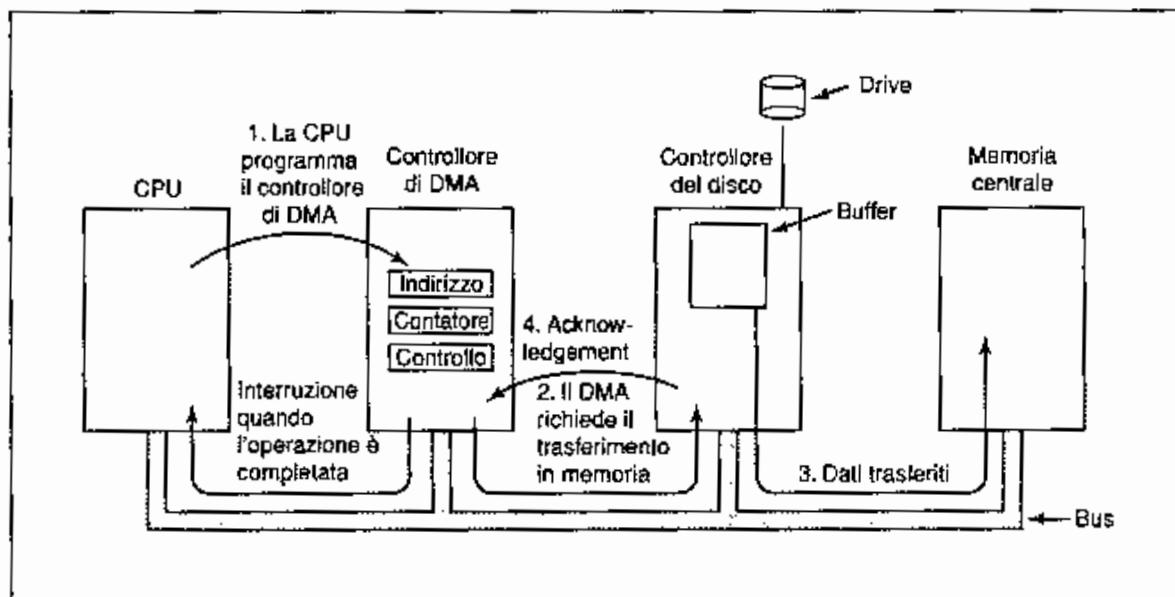


Figura 5.4 Un'operazione di trasferimento DMA.

Il controllore del DMA inizia il trasferimento emettendo una richiesta di lettura sul bus verso il controllore del disco (passo 2); questa richiesta ha l'aspetto di una qualunque altra richiesta di lettura, e il controllore del disco non sa e non gli importa sapere se proviene dalla CPU o dal controllore del DMA. Solitamente, l'indirizzo di memoria nel quale scrivere si trova sulle linee di indirizzamento del bus, cosicché, quando il controllore del disco porta la parola successiva dal suo buffer interno, sa dove copiarla. La scrittura in memoria è un altro ciclo di bus standard (passo 3). Quando la scrittura è stata completata, il controllore del disco invia un segnale di ricevuta (acknowledgement) al controllore del disco (passo 4). Successivamente, il controllore del DMA incrementa l'indirizzo di memoria da usare e decremente il contatore di byte: vengono ripetuti i passi dal 2 al 4 finché il contatore di byte non è uguale a 0. A quel punto il controllore di DMA invia un'interruzione alla CPU per farle sapere che il trasferimento è completo. Quando il sistema operativo viene avviato, non ha bisogno di copiare il blocco di disco in memoria, perché è già lì.

I controllori di DMA variano considerevolmente di complessità: i più semplici gestiscono un trasferimento alla volta, come appena descritto, mentre i più complessi possono essere programmati per effettuare contemporaneamente trasferimenti multipli. Questi ultimi hanno insieme multipli di registri, uno per ogni canale: la CPU inizia con il caricare ogni insieme di registri con i parametri necessari al trasferimento ad esso relativo; ogni trasferimento deve riguardare un diverso controllore di dispositivo, e dopo che ogni parola è trasferita (dal passo 2 al passo 4 in Figura 5.4) il controllore di DMA decide quale dispositivo servire. Esso può essere configurato per servire le richieste con un algoritmo round-robin, o avere uno schema di priorità che favorisce alcuni dispositivi rispetto ad altri. Richieste multiple a diversi controllori di dispositivo possono rimanere in sospeso nello stesso momento, ammesso che vi siano dei modi non ambigui per gestire i messaggi di acknowledgement; spesso, per questo motivo, si usa una diversa linea di acknowledgement per ogni canale DMA.

Molti bus possono operare in due modi: in modo "una parola alla volta" oppure in modo blocco; anche alcuni controllori DMA possono operare in entrambi i modi. Nel primo modo, l'operazione è esattamente come è stata descritta in precedenza: il controllore di DMA richiede il trasferimento di una parola e prende possesso del bus, e se anche la CPU ha bisogno del bus, essa deve aspettare. Il meccanismo è detto di **furto del ciclo** (cycle stealing) perché il controllore del dispositivo si intrufola e ruba un ciclo di bus ogni tanto alla CPU, ritardandola leggermente. Nel modo blocco, il controllore di DMA dice al dispositivo di acquisire il bus, produce una serie di trasferimenti e quindi rilascia il bus. Questa operazione, detta **modo burst**, è più efficiente del furto del ciclo in quanto acquisire il bus richiede tempo, e in questo modo viene trasferita più di una parola, al costo di una singola acquisizione di bus. Il lato negativo del modo burst è che può bloccare la CPU e altri dispositivi per un lungo periodo di tempo, nel caso di trasferimenti grandi.

Nel modello che abbiamo discusso finora, che talvolta è detto **modo fly-by** (al volo), il controllore di DMA dice al controllore di dispositivo di trasferire i dati direttamente in memoria centrale; un modo alternativo, che usano alcuni controllori di DMA, è quello in cui il controllore di dispositivo invia la parola al controllore di DMA, il quale emette una seconda richiesta di bus per poter scrivere la parola nel posto giusto. Questo schema richiede un ulteriore ciclo di bus per ogni parola trasferita, ma è più flessibile, in quanto può anche effettuare copie tra dispositivi, e anche copie tra memoria e memoria (emettendo prima una richiesta di lettura alla memoria e poi una richiesta di scrittura ad un diverso indirizzo di memoria).

La maggior parte dei controllori di DMA usa indirizzi di memoria fisica per i trasferimenti; questo obbliga il sistema operativo a dover convertire l'indirizzo virtuale del bu-

fer di memoria richiesto in un indirizzo fisico, e scrivere tale indirizzo fisico nel registro degli indirizzi del controllore di DMA. Uno schema alternativo usato da alcuni controllori di DMA è quello di scrivere nel registro del controllore di DMA l'indirizzo virtuale, e fare in modo che il controllore di DMA usi la MMU per la conversione da virtuale a fisico. Solo nel caso in cui la MMU sia parte della memoria (cosa possibile ma rara) piuttosto che della CPU, gli indirizzi virtuali possono essere messi sul bus.

In precedenza abbiamo menzionato il fatto che il disco deve leggere i dati nel suo buffer interno prima che il DMA possa iniziare; vi potrete chiedere come mai il controllore non immagazzina semplicemente i byte in memoria centrale appena li ottiene dal disco, ossia perché abbia bisogno di un buffer interno. Le ragioni sono due: la prima è che usando il buffer interno, il controllore del disco può verificare il checksum prima di iniziare il trasferimento; se il checksum non è corretto si segnala l'errore e non si fa nessun trasferimento.

La seconda ragione è che nel momento in cui il trasferimento inizia, i bit arrivano dal disco ad una frequenza costante, sia che il controllore sia pronto a riceverli sia che non lo sia; se il controllore cercasse di scrivere direttamente in memoria, dovrebbe usare il bus di sistema per ogni parola trasferita, e se il bus fosse occupato da qualche altro dispositivo (ad esempio in modo burst), il controllore dovrebbe aspettare. A questo punto se una parola arrivasse dal disco prima di avere immagazzinato le precedenti, il controllore dovrebbe metterla da qualche altra parte. Se il bus fosse molto occupato, il controllore potrebbe salvare un certo numero di parole e avere un bel po' di amministrazione da fare. Se invece il blocco è salvato in un buffer interno, il bus non viene utilizzato finché il DMA non inizia, quindi la progettazione del controllore è molto più semplice in quanto il trasferimento in memoria, dal DMA, non è critico dal punto di vista temporale (alcuni vecchi controllori, in effetti, andavano direttamente in memoria con molto poco buffering interno, ma quando il bus era molto occupato poteva succedere che un trasferimento terminasse con un errore di esecuzione).

Non tutti i computer usano il DMA; contro di esso si può dire che la CPU centrale è spesso molto più veloce di un controllore DMA e può lavorare molto più velocemente (quando non si ha il fattore limitante della velocità del dispositivo di I/O). Nel caso non abbia altro da fare, il fatto che la CPU (veloce) debba aspettare che il controllore di DMA (lento) finisca sembra inutile. Inoltre, eliminare il controllore di DMA e fare sì che la CPU faccia tutto il lavoro via software fa risparmiare denaro, e questo è importante nel caso di computer embedded di fascia bassa.

5.1.5 Le interruzioni rivisitate

Abbiamo introdotto brevemente le interruzioni nella Sezione 1.4.3, ma ci sono molte altre cose da dire. In un tipico sistema di un personal computer, la struttura ad interruzioni è come quella mostrata in Figura 5.5. A livello hardware le interruzioni funzionano nel modo seguente: quando un dispositivo di I/O termina il lavoro che gli è stato assegnato, causa un'interruzione (supponendo che le interruzioni siano state abilitate dal sistema operativo), impostando un segnale su una linea di bus che gli è stata assegnata. Tale segnale è rilevato dal controllore delle interruzioni sulla scheda madre, il quale decide cosa fare.

Se non vi sono altre interruzioni sospese, il controllore di interruzioni gestisce l'interruzione immediatamente; mentre se vi è un'interruzione in corso, oppure se un altro dispositivo ha emesso una richiesta di interruzione simultanea su una linea di richiesta interruzione a più alta priorità, il dispositivo viene momentaneamente ignorato; esso continua ad impostare il segnale di interruzione finché non viene servito dalla CPU.

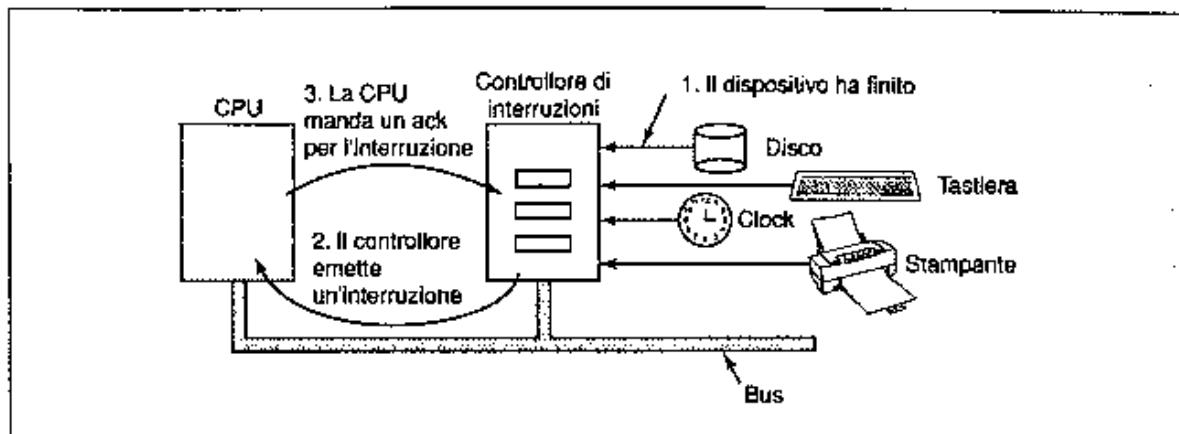


Figura 5.6 Un'interruzione si verifica in questo modo. In realtà le connessioni tra i dispositivi e il controllore delle interruzioni usano linee di interruzione sul bus, piuttosto che cavi dedicati.

Per gestire l'interruzione, il controllore mette sulle linee di indirizzamento un numero, che specifica quale dispositivo vuole richiamare l'attenzione, ed imposta un segnale di interruzione per la CPU.

In seguito al segnale di interruzione, la CPU interrompe l'operazione corrente, ed inizia a fare qualcos'altro. Il numero mandato sulle linee di indirizzamento viene usato come un indice in una tabella chiamata *vettore delle interruzioni* per cercare un nuovo contatore di programma (PC, Program Counter). Questo contatore punta all'inizio della procedura di inizio interruzione corrispondente; tipicamente le operazioni di trap e le interruzioni usano lo stesso meccanismo, e spesso lo stesso vettore delle interruzioni. La posizione del vettore può essere cablata nella macchina, o si può metterla in un qualsiasi punto della memoria, con un registro della CPU (caricato dal sistema operativo) che punta al suo inizio.

Subito dopo l'inizio della sua esecuzione, la procedura di servizio delle interruzioni invia un acknowledgement all'interruzione, scrivendo un certo valore su una delle porte di I/O del controllore delle interruzioni; questo messaggio di acknowledgement serve al controllore per sapere quando è libero di emettere un'altra interruzione. Permettendo alla CPU di ritardare l'acknowledgement finché non è pronta a gestire altre interruzioni, si possono evitare condizioni di corsa critica riguardanti interruzioni multiple e quasi simultanee. Alcuni computer più vecchi, però, non sono dotati di un controllore delle interruzioni centralizzato, quindi ogni controllore di dispositivo richiede le proprie interruzioni.

L'hardware salva sempre un certo numero di informazioni prima di iniziare la procedura di servizio. Quali informazioni vengono salvate e come vengono salvate varia enormemente a seconda della CPU usata; come minimo, occorre salvare il contatore di programma, in modo tale che un processo interrotto possa essere ripreso; all'altro estremo, possono essere salvati tutti i registri visibili e un grande numero di registri interni.

Un problema è dove salvare questo genere di informazioni; una possibilità è quella di metterle in registri interni che possano essere letti dal sistema operativo, ma in questo modo il controllore delle interruzioni non può ricevere l'acknowledgement finché tutte le informazioni potenzialmente importanti non sono state lette, se si vuole evitare una seconda interruzione che potrebbe sovrascrivere i registri interni che salvano lo stato. Questa strategia porta a lunghi tempi morti, in cui le interruzioni vengono disabilitate, e forse a interruzioni perse e dati persi.

Di conseguenza, la maggior parte delle CPU salvano le informazioni in una pila, ma anche questo approccio crea dei problemi. Prima di tutto: la pila di chi? Se viene usata quella corrente, potrebbe per esempio essere la pila di un processo utente. Il puntatore alla pila potrebbe addirittura non essere legale, e questo provocherebbe un errore fatale se l'hardware cercasse di scrivere una parola dove esso punta. Inoltre potrebbe anche puntare alla fine di una pagina, e dopo una serie di scritture in memoria, si potrebbe superare il limite della pagina e generare un fault di pagina (page fault); in questo caso, ossia se si ha un fault di pagina durante l'elaborazione di un'interruzione hardware, ci si trova davanti ad un problema anche più grande: dove salvare lo stato prima di gestire il fault di pagina?

Se si usa la pila del kernel è più probabile che il puntatore alla pila sia legale, e che punti ad una pagina caricata; però passare in modo kernel può richiedere un cambiamento di contesto della MMU e probabilmente invalidare gran parte della cache e del TLB. Ricaricarle aumenterà il tempo di elaborazione dell'interruzione, sia staticamente sia dinamicamente, e sprecherà il tempo di CPU.

Un altro problema è causato dal fatto che la maggior parte delle moderne CPU sono fortemente basate su pipeline, e spesso sono superscalari (ossia internamente parallele). Nei sistemi più vecchi, dopo la fine dell'esecuzione di ogni istruzione, il microprogramma o l'hardware controllavano l'eventuale presenza di interruzioni pendenti; se questo accadeva, il contatore di programma e il PSW venivano messi sulla pila, e la sequenza di interruzione aveva inizio. Alla fine dell'esecuzione del gestore delle interruzioni, si iniziava il processo opposto, ossia il vecchio PSW e il contatore di programma venivano tolti dalla pila, e il processo precedente ripartiva.

Questo modello suppone implicitamente che, se l'interruzione avviene dopo una qualche istruzione, tutte le istruzioni precedenti ad essa, essa inclusa, siano state eseguite completamente, e nessuna istruzione dopo di essa sia stata eseguita. Su macchine di vecchia generazione, queste ipotesi erano sempre valide, ma sulle macchine odierne questo può non essere vero.

Per cominciare, si consideri il modello di pipeline di Figura 1.6(a); cosa succede se avviene un'interruzione quando la pipeline è piena (come solitamente accade)? Ci sono molte istruzioni a svariati livelli di esecuzione e, quando capita l'interruzione, il valore del contatore di programma può non riflettere il valore corretto che separa le istruzioni eseguite da quelle non eseguite; più probabilmente avrà l'indirizzo dell'istruzione successiva che deve essere recuperata e messa nella pipeline, piuttosto che l'indirizzo dell'istruzione che è appena stata elaborata dall'unità di esecuzione.

Di conseguenza, ci può essere un limite molto ben definito tra istruzioni che sono state effettivamente eseguite e altre che non sono state eseguite, ma l'hardware può non esserne a conoscenza. Quindi, quando il sistema operativo deve riprendere dopo un'interruzione, non può semplicemente iniziare a riempire la pipeline a partire dall'indirizzo contenuto nel contatore di programma, ma deve capire qual era l'ultima istruzione eseguita, e questo è spesso un compito complicato che può richiedere l'analisi dello stato della macchina.

Anche se questa situazione non è buona, le interruzioni su una macchina superscalare, come quella di Figura 1.6(b) sono molto peggio; siccome in questo caso le istruzioni possono venire eseguite non in ordine, può non esserci una chiara distinzione tra istruzioni eseguite ed istruzioni non eseguite. Può succedere, ad esempio, che le istruzioni 1, 2, 3, 5 e 8 siano state eseguite, mentre le istruzioni 4, 6, 7, 9, 10 e successive non lo siano; inoltre il contatore di programma potrebbe puntare all'istruzione 9, o alla 10, o alla 11.

Un'interruzione che lascia la macchina in uno stato ben definito è detta **interruzione precisa** (Walzer e Cragon, 1995) ed ha le seguenti proprietà:

1. Il PC (Program Counter, contatore di programma) è salvato in un posto noto.
2. Tutte le istruzioni precedenti a quella puntata dal PC sono state eseguite completamente.
3. Nessuna istruzione successiva a quella puntata dal PC è stata eseguita.
4. Lo stato di esecuzione dell'istruzione puntata dal PC è noto.

Notare che non si impedisce l'inizio dell'esecuzione delle istruzioni successive a quella puntata dal PC, ma ogni cambiamento che esse possono aver apportato ai registri o alla memoria deve essere annullato prima dell'interruzione. È permesso che l'istruzione puntata dal PC sia stata completamente eseguita, e anche che non lo sia stata, però deve essere chiaro quale dei due casi sia vero. Spesso, se l'interruzione è di I/O, l'istruzione non sarà iniziata, però se l'interruzione è un'operazione di trap o un fault di pagina, il PC punta solitamente all'istruzione che l'ha causata, cosicché in seguito possa ricominciare da dove era stato interrotto.

Un'interruzione che non abbia queste caratteristiche è detta **interruzione imprecisa**, e rende la vita estremamente difficile al progettista di sistemi operativi, che deve capire cosa sta succedendo e quello che deve ancora succedere. Le macchine con interruzioni imprecise solitamente riversano una gran quantità di stati interni sulla pila, per dare al sistema operativo la possibilità di capire cosa stia succedendo. Il fatto di dover salvare grandi quantità di informazioni in memoria ad ogni interruzione, rende le interruzioni lente e i ripristini anche più lenti. Questo porta all'ironica situazione di avere delle CPU superscalari molto veloci, che talvolta non sono adatte a lavorare in tempo reale a causa del fatto che la gestione delle interruzioni è lenta.

Alcuni computer sono progettati in modo tale che alcune interruzioni e operazioni di trap siano precise e altre no; ad esempio, avere interruzioni di I/O precise, e trap dovute a errori fatali nei programmi imprecise non è grave, in quanto non si dovranno fare tentativi di ripartire il processo in esecuzione. Alcune macchine hanno un bit che può essere impostato per fare in modo che tutte le interruzioni diventino precise; il lato negativo dell'impostare questo bit è che essa forza la CPU a tener traccia di ogni cosa che stia facendo, e a mantenere copie dei registri così che possa generare interruzioni precise in ogni istante; tutto questo overhead è di grande impatto sulle prestazioni.

Alcune macchine superscalari, quali il Pentium Pro e i suoi successori, hanno interruzioni precise per permettere a vecchi programmi per 386, 486 e Pentium I di funzionare in modo corretto (il primo modello di Pentium superscalare è il Pro, mentre il Pentium I aveva solo due pipeline). Il prezzo pagato per avere interruzioni precise è una logica delle interruzioni all'interno della CPU estremamente complessa, per fare sì che quando il controllore delle interruzioni vuole segnalare un'interruzione, tutte le istruzioni fino ad un certo punto possano terminare, mentre da quel punto in poi nessuna istruzione possa produrre effetti sullo stato della macchina. In questo caso il prezzo è pagato non in termini di tempo, ma in area del circuito integrato, e in complessità di progettazione. Se non fossero necessarie interruzioni precise per motivi di compatibilità, questa area sul circuito integrato potrebbe essere utilizzata per cache più larghe, rendendo la CPU più veloce. D'altro canto, interruzioni imprecise rendono il sistema operativo troppo lento e complicato, quindi diventa difficile capire quale dei due approcci sia il migliore.

5.2 Principi del software di I/O

Si lasci ora da parte l'hardware e si prenda in considerazione la struttura del software di I/O. Prima di tutto analizzeremo gli scopi del software di I/O ed in seguito i diversi modi in cui si può fare I/O dal punto di vista del sistema operativo.

5.2.1 Scopi del software di I/O

Un concetto chiave nel progetto del software di I/O è l'**indipendenza dal dispositivo fisico**; ciò significa che dovrebbe essere possibile scrivere programmi dai quali si possa accedere ad ogni dispositivo di I/O, senza dover specificare in anticipo il dispositivo stesso; per esempio, un programma che in input legge un file dovrebbe essere in grado di usare un file su dischetto, su disco rigido o su CD-ROM, senza dover modificare il codice per ogni tipo di dispositivo. In modo simile, si dovrebbe essere in grado di battere il comando

```
sort <input>output
```

e vederlo funzionare con l'input proveniente dal dischetto, da disco rigido IDE o SCSI, o da tastiera, e l'output diretto verso ogni tipo di disco o verso il monitor. Dipende dal sistema operativo prendersi cura dei problemi causati dal fatto che questi dispositivi possono essere realmente differenti, e che richiedono sequenze di comandi molto diverse per leggere e scrivere.

Strettamente correlato all'indipendenza dal dispositivo fisico, è il problema della **denominazione uniforme** (uniform naming). Il nome di un file o di un dispositivo, dovrebbe essere semplicemente una stringa di caratteri o un numero intero, e non dovrebbe in alcuna maniera dipendere dal dispositivo. In UNIX, tutti i dischi si possono montare sul file system gerarchico in posti arbitrari, cosicché l'utente non deve necessariamente sapere quale dispositivo corrisponda ad un certo nome. Per esempio, un dischetto si può montare all'inizio della directory `/usr/ast/backup` in modo che copiando un file in `/usr/ast/backup/lunedì` si copia il file sul dischetto. In questo modo, tutti i file ed i dispositivi sono indirizzati alla stessa maniera, utilizzando un path name.

Un altro concetto importante del software di I/O è il trattamento degli errori. In genere, gli errori dovrebbero essere trattati il più vicino possibile all'hardware. Se il controllore scopre un errore di lettura, se può, dovrebbe cercare di correggerlo egli stesso; se non fosse possibile correggerlo, allora il driver di dispositivo dovrebbe gestirlo, magari semplicemente ritentando la lettura del blocco. Molti errori sono transitori, come per esempio gli errori in lettura provocati da particelle di polvere sulla testina di lettura, e non si ripresentano se l'operazione viene ripetuta. I livelli alti dovrebbero essere avvertiti della presenza dell'errore solamente nel caso in cui i livelli più bassi non fossero in grado di risolvere il problema; ma in molti casi, il recupero dell'errore si può realizzare ai livelli bassi in modo trasparente, senza che i livelli superiori se ne accorgano.

Un altro concetto chiave è il trasferimento sincrono piuttosto che asincrono (guidato dall'interruzione). La maggior parte dell'I/O è asincrono: la CPU fa partire il trasferimento e passa a fare qualcos'altro, finché non arriva l'interruzione; ma i programmi utente sono più facili da scrivere se le operazioni di I/O sono sincrone: dopo un comando di READ, il programma viene automaticamente sospeso finché i dati non sono disponibili nel buffer. Sta al sistema operativo, rendere sincrone agli occhi dei programmi utente le operazioni che sono intrinsecamente guidate dalle interruzioni, e cioè asincrone.

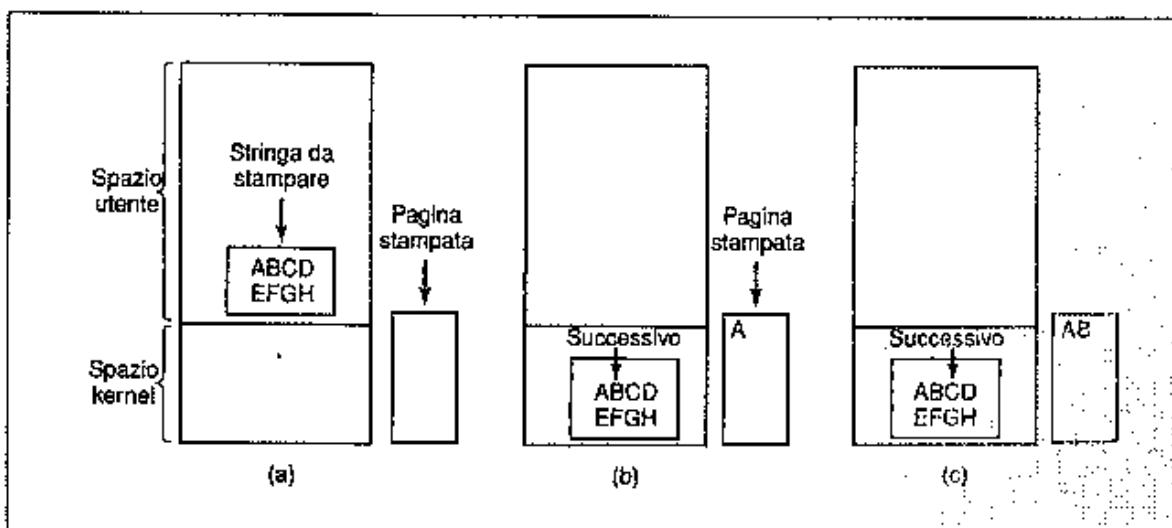
Un altro concetto riguardante il software di I/O è quello di **bufferizzazione**. Spesso i dati che escono da un dispositivo non possono essere portati immediatamente nella loro destinazione finale; per esempio, quando un pacchetto arriva dalla rete, il sistema operativo non sa dove metterlo finché non lo ha immagazzinato da qualche parte e lo ha esaminato. Inoltre alcuni dispositivi hanno dei vincoli di tempo reale molto severi (ad esempio dispositivi audio digitali), quindi i dati devono essere messi in un buffer di output in anticipo, per distinguere la frequenza con cui il buffer è riempito da quella con cui è svuotato, allo scopo di evitare lo svuotamento del buffer. La bufferizzazione richiede molte operazioni di copiatura, e quindi ha spesso un grande impatto sulle prestazioni dell'I/O.

Il concetto finale che si tratterà riguarda i dispositivi dedicati e quelli condivisibili. Alcuni dispositivi di I/O, come i dischi, possono essere usati da più utenti contemporaneamente; il fatto di avere più utenti con file aperti sullo stesso disco e nello stesso momento non causa problemi. Altri dispositivi, come i dispositivi a nastro, devono essere dedicati ad un singolo utente finché quell'utente non abbia finito; a questo punto un altro utente può usare il dispositivo. Se si avessero due o più utenti che scrivono blocchi mischiati a caso sullo stesso nastro non si otterrebbero buoni risultati. L'introduzione di dispositivi dedicati (non condivisi) genera anche tutta una serie di problemi, tra i quali situazioni di stallo; è ancora il sistema operativo che deve trattare sia i dispositivi dedicati, sia quelli condivisibili, in maniera tale da evitare problemi.

5.2.2 I/O programmato

Ci sono fondamentalmente tre modi diversi in cui l'I/O possa essere effettuato; in questa sezione analizzeremo il primo (l'I/O programmato), mentre nelle prossime due esamineremo gli altri (l'I/O guidato dal dispositivo e l'I/O tramite DMA). La forma più semplice di I/O è quella che lascia alla CPU il compito di fare tutto il lavoro, questo metodo è chiamato **I/O programmato**.

È più facile spiegare l'I/O programmato attraverso un esempio; si consideri quindi un processo utente che voglia stampare una stringa di otto caratteri "ABCDEFGH" sulla stampante. Prima di tutto esso assemblerà la stringa in un buffer dello spazio utente, come mostrato in Figura 5.6(a).



A questo punto il processo utente acquisisce la stampante in scrittura, tramite una chiamata di sistema; se la stampante è usata da un altro processo, questa chiamata fallirà e ritornerà un codice di errore o si bloccherà finché la stampante non diventa disponibile, a seconda del sistema operativo e dei parametri della chiamata di sistema. Nel momento in cui ha la stampante, il processo utente fa una chiamata di sistema per dire al sistema operativo di stampare la stringa sulla stampante.

Il sistema operativo (solitamente) copia il buffer contenente la stringa in un array dello spazio kernel, sia esso *p*, dove sia più facile accedervi, dato che il kernel può dover cambiare la mappatura della memoria per accedere allo spazio utente. A questo punto il sistema operativo controlla per vedere se la stampante è libera. Se non lo è aspetta, e nel momento in cui questa viene liberata, il sistema operativo copia il primo carattere nel registro dei dati della stampante, utilizzando ad esempio l'I/O mappato in memoria, e questa azione attiva la stampante. Il carattere può non apparire subito perché molte stampanti aspettano di avere una linea o una pagina nel proprio buffer prima di iniziare a stampare; in Figura 5.6(b), comunque, vediamo che il primo carattere è stato stampato e che il sistema ha marcato la "B" come il successivo carattere da stampare.

Appena il primo carattere è stato copiato sulla stampante, il sistema operativo controlla per vedere se la stampante è pronta a riceverne un altro. Generalmente la stampante ha un secondo registro che contiene lo stato della stampante stessa; l'azione di scrivere nel registro dei dati porta la stampante in uno stato di non pronto, mentre quando il controllore della stampante ha elaborato il carattere corrente, essa indica la propria disponibilità impostando qualche bit nel suo registro dello stato, o inserendo qualche apposito valore all'interno di esso.

A questo punto il sistema operativo aspetta che la stampante sia pronta, e poi stampa il carattere successivo, come mostrato in Figura 5.6(c). Questo ciclo continua finché non si è stampata l'intera stringa; a questo punto il controllo ritorna al processo utente.

Le azioni eseguite dal sistema operativo sono riassunte in Figura 5.7: prima i dati sono copiati nel kernel, poi il sistema operativo entra in un ciclo emettendo i caratteri uno alla volta. L'aspetto essenziale dell'I/O programmato, mostrato chiaramente in questa figura, è che dopo l'emissione di un carattere, la CPU controlla continuamente il dispositivo per vedere se è pronto ad accettare un altro carattere. Questo comportamento è chiamato polling o attesa attiva (busy waiting).

L'I/O programmato è semplice, ma ha lo svantaggio di legare la CPU a tempo pieno fino al completamento dell'operazione di I/O. Se il tempo necessario a stampare il carattere è breve (perché quello che la stampante fa è solo copiare il nuovo carattere in un buffer interno), allora l'attesa attiva è accettabile e, in generale, in sistemi embedded, dove la CPU non ha altro da fare, l'attesa attiva è ragionevole. Però, in sistemi più complessi, nei quali la CPU sia più occupata, l'attesa attiva è inefficiente e si rende necessario un metodo di I/O migliore.

```
copia_da_utente(buffer,p,contatore)
for(i = 0; i < contatore; i++) {
    while(*stato_stampante != PRONTO)
        *registro_dati_stampante = p[i];
}
ritorna_a_utente();
```

/*p è il buffer del kernel */
/*ciclo su ogni carattere*/
/*attendi finché non pronto*/
/*un carattere in output*/

Figura 5.7 Operazione di scrittura di una stringa sulla stampante tramite I/O programmato.

5.2.3 I/O guidato dalle interruzioni

Ora consideriamo il caso in cui una stampante non metta i caratteri all'interno di un buffer, ma li stampi man mano che arrivano. Se per esempio una stampante può stampare 100 caratteri al secondo, la stampa di ogni carattere richiede 10 ms; questo significa che dopo che ogni carattere è stato scritto nel registro di dati della stampante, la CPU rimane inattiva per 10 ms in attesa che le sia permesso di emettere il carattere successivo. Questo intervallo di tempo sarebbe più che sufficiente per fare un cambio di contesto ed eseguire qualche altro processo, nei 10 ms che altrimenti andrebbero sprecati.

Il modo di permettere alla CPU di fare qualcosa' altro mentre sta aspettando che la stampante sia pronta a ricevere il carattere successivo è quello di utilizzare il meccanismo delle interruzioni. Quando viene fatta la chiamata di sistema per la stampa della stringa, il buffer viene copiato nello spazio del kernel, come descritto in precedenza, e il primo carattere è copiato verso la stampante appena questa è pronta a riceverlo; a questo punto la CPU chiama lo scheduler e qualche altro processo viene scelto per l'esecuzione, mentre il processo che ha richiesto la stampa della stringa è bloccato finché l'intera stringa non è stata stampata. Il lavoro fatto dalla chiamata di sistema è descritto in Figura 5.8(a).

```

copia_da_utente(buffer, p, contatore);      if (contatore == 0) {
abilita_interruzioni();                      sblocca_utente();
while(*stato_stampante != PRONTO)            } else {
*registro_dati_stampante = p[0];              *registro_dati_stampante = p[i];
scheduler();                                    contatore = contatore - 1;
                                                i = i + 1;
}
acknowledge_interruzione();
ritorna_da_interruzione();

(a)                                            (b)

```

Figura 5.8 Scrittura di una stringa sulla stampante tramite I/O guidato dalle interruzioni. (a) Codice eseguito nel momento in cui viene eseguita la chiamata di sistema di stampa. (b) Procedura di servizio dell'interruzione.

Quando la stampante ha stampato il carattere ed è pronta a ricevere il successivo, genera un'interruzione, che ferma il processo corrente e salva il suo stato; a questo punto viene eseguita la procedura di servizio delle interruzioni della stampante. Una versione rudimentale di questo codice è mostrata in Figura 5.8(b): se non ci sono altri caratteri da stampare, il driver delle interruzioni si occupa di sbloccare l'utente, altrimenti emette il carattere successivo e poi ritorna al processo che era in esecuzione prima dell'interruzione, continuando da dove aveva interrotto.

5.2.4 I/O tramite DMA

Un ovvio svantaggio dell'I/O guidato dalle interruzioni è che richiede un'interruzione per ogni carattere, e le interruzioni richiedono tempo, quindi questo schema spreca una certa quantità di tempo della CPU. Una soluzione è quella di usare il DMA; l'idea è quella che il controllore del DMA emetta i caratteri e li passi alla stampante uno alla volta.

```

copia_da_utente(buffer, p, contatore);
imposta_controllore_DMA();
scheduler();                                acknowledge_interruzione();
                                                sblocca_utente();
                                                ritorna_da_interruzione();

```

(a)

(b)

Figura 5.9 Stampa di una stringa tramite il DMA. (a) Codice eseguito quando viene effettuata la chiamata di sistema di stampa. (b) Procedura di gestione dell'interruzione.

ta, senza disturbare la CPU. In essenza, il DMA è I/O programmato, con la differenza che è il controllore del DMA a fare tutto il lavoro invece della CPU principale. Uno schema del codice è dato in Figura 5.9.

Il guadagno che si ottiene usando il DMA è quello di ridurre il numero di interruzioni, da una per ogni carattere da stampare a una per ogni buffer stampato: se ci sono molti caratteri e le interruzioni sono lente questo può essere un miglioramento sostanziale. D'altro canto, il controllore di DMA è solitamente molto più lento della CPU principale e, se il controllore di DMA non è in grado di far andare il dispositivo a velocità massima, o se la CPU non ha molto da fare mentre sta aspettando l'interruzione dal DMA, allora l'I/O guidato dalle interruzioni o addirittura l'I/O programmato possono essere scelte migliori.

5.3 I livelli del software di I/O

Il software di I/O è tipicamente organizzato in quattro livelli, come mostrato in Figura 5.10; ogni livello ha una funzione ben definita e un'interfaccia verso i livelli contigui ben definita. Le funzionalità e le interfacce cambiano da sistema a sistema: la discussione che segue, che esamina tutti i livelli a partire dal basso, non è specifica di nessuna macchina in particolare.

5.3.1 Driver delle interruzioni

Mentre l'I/O programmato può essere utile, in genere le interruzioni di I/O sono una realtà spiacevole che però non può essere evitata. Esse dovrebbero essere nascoste nelle viscere del sistema in modo che ne sia a conoscenza la minor parte possibile del sistema stesso. La maniera migliore per nasconderle è quella di far sì che il driver di un dispositivo si blocchi ogni volta che da' un comando di I/O, aspettando la fine dell'I/O e l'interruzione. Per esempio, il processo può bloccarsi utilizzando un comando `down` su un semaforo, un comando `wait` su una variabile di condizione, un comando `receive` di un messaggio, o qualcosa del genere.

Quando l'interruzione avviene, la procedura che la serve fa tutto ciò che è necessario per gestire l'interruzione, e sbloccherà il processo che l'ha fatta partire: in alcuni sistemi, manderà un segnale `up` su di un semaforo; in altri farà una `signal` su una variabile di condizione di un qualche monitor; in altri sistemi ancora, spedirà un messaggio al processo bloccato. In tutti i casi, l'effetto globale di un'interruzione sarà che un processo che era stato precedentemente bloccato, sarà in grado di riprendere l'esecuzione. Questo

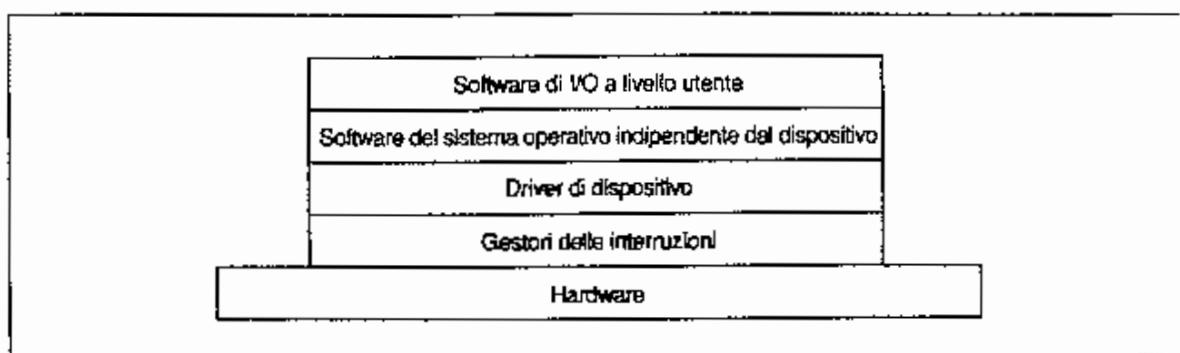


Figura 5.10 I livelli del sistema di software di I/O.

modello funziona meglio per i driver che sono strutturati come processi del kernel, e possiedono il loro stato, le loro pile ed i loro contatori di programma.

Ovviamente la realtà non è così semplice, ed elaborare un'interruzione non è semplicemente questione di prendere l'interruzione, mandare un up su un semaforo ed in seguito eseguire una RET in modo da tornare al processo precedente: il sistema operativo ha molto più lavoro da fare. A questo punto daremo un'idea di quale sia questo lavoro descrivendolo come una serie di passi che devono essere eseguiti in software, dopo che l'interruzione hardware è stata eseguita. Occorre notare che i dettagli dipendono molto dal sistema, quindi alcuni passi descritti nel seguito possono non essere necessari su una macchina specifica, mentre altri non inclusi nella lista possono essere richiesti; inoltre, i passi possono essere eseguiti in ordine diverso a seconda della macchina in questione:

1. Salvare i registri (incluso il PSW) che non sono ancora stati salvati dall'hardware di interruzione.
2. Impostare un contesto per la procedura di servizio dell'interruzione; questo può richiedere di impostare una TLB, una MMU, e una tabella delle pagine.
3. Organizzare una pila per la procedura di servizio delle interruzioni.
4. Mandare un acknowledgement al controllore delle interruzioni; se non c'è un controllore centralizzato, riabilitare le interruzioni.
5. Copiare i registri dal luogo in cui erano stati salvati (solitamente qualche pila) nella tabella dei processi.
6. Eseguire la procedura di servizio delle interruzioni, che estrarrà le informazioni dai registri del controllore del dispositivo che ha causato l'interruzione.
7. Scegliere quale processo eseguire; se l'interruzione ha sbloccato qualche processo ad alta priorità, esso diventa un buon candidato.
8. Organizzare un contesto di MMU per il processo successivo; può anche essere necessaria una TLB.
9. Caricare i registri del nuovo processo, incluso il suo PSW.
10. Incominciare l'esecuzione del nuovo processo.

Come si può notare, l'esecuzione di un'interruzione è tutto tranne che facile; tra le altre cose richiede una considerevole quantità di istruzioni della CPU, specialmente su macchine con memoria virtuale e dove le tabelle delle pagine devono essere organizzate, o lo stato della MMU deve essere memorizzato (ad esempio, i bit R e M). Su alcune macchine la TLB e la cache di CPU possono richiedere una gestione, durante il passaggio da modo utente a modo kernel, il che richiede ulteriori cicli macchina.

5.3.2 Driver dei dispositivi

Nelle pagine precedenti si è visto cosa fanno i controllori delle periferiche (dispositivi di I/O); si è visto anche che ogni controllore ha uno o più registri che servono a spedirgli i comandi, oppure alcuni registri che servono per conoscere lo stato del controllore, o entrambi. Il numero di registri del dispositivo e la natura dei comandi varia radicalmente da dispositivo a dispositivo; per esempio il driver di un mouse deve accettare informazioni dal mouse relative a quanto si è mosso e quali tasti sono stati premuti, mentre un driver di disco deve ricevere informazioni circa i settori, le tracce, i cilindri, le testine, il movimento del braccio meccanico, gli attuatori del motore, i tempi di posizionamento della testina e tutti i dettagli della meccanica che sono necessari a far funzionare correttamente il disco. Chiaramente, i due driver saranno molto diversi.

Come conseguenza, ogni dispositivo di I/O collegato ad un computer ha bisogno di codice specifico che controlli il dispositivo stesso. Questo codice è detto **driver del dispositivo** ed è generalmente scritto dal produttore del dispositivo e consegnato insieme con il dispositivo stesso; siccome ogni sistema operativo ha bisogno dei propri driver, i produttori solitamente forniscono driver per i sistemi operativi più comuni.

Ogni driver di dispositivo solitamente controlla un solo tipo, o, al più, una classe di dispositivi strettamente correlati; per esempio il driver di un disco SCSI può solitamente gestire dischi SCSI diversi, di dimensioni e velocità diverse e forse anche CD-ROM SCSI.

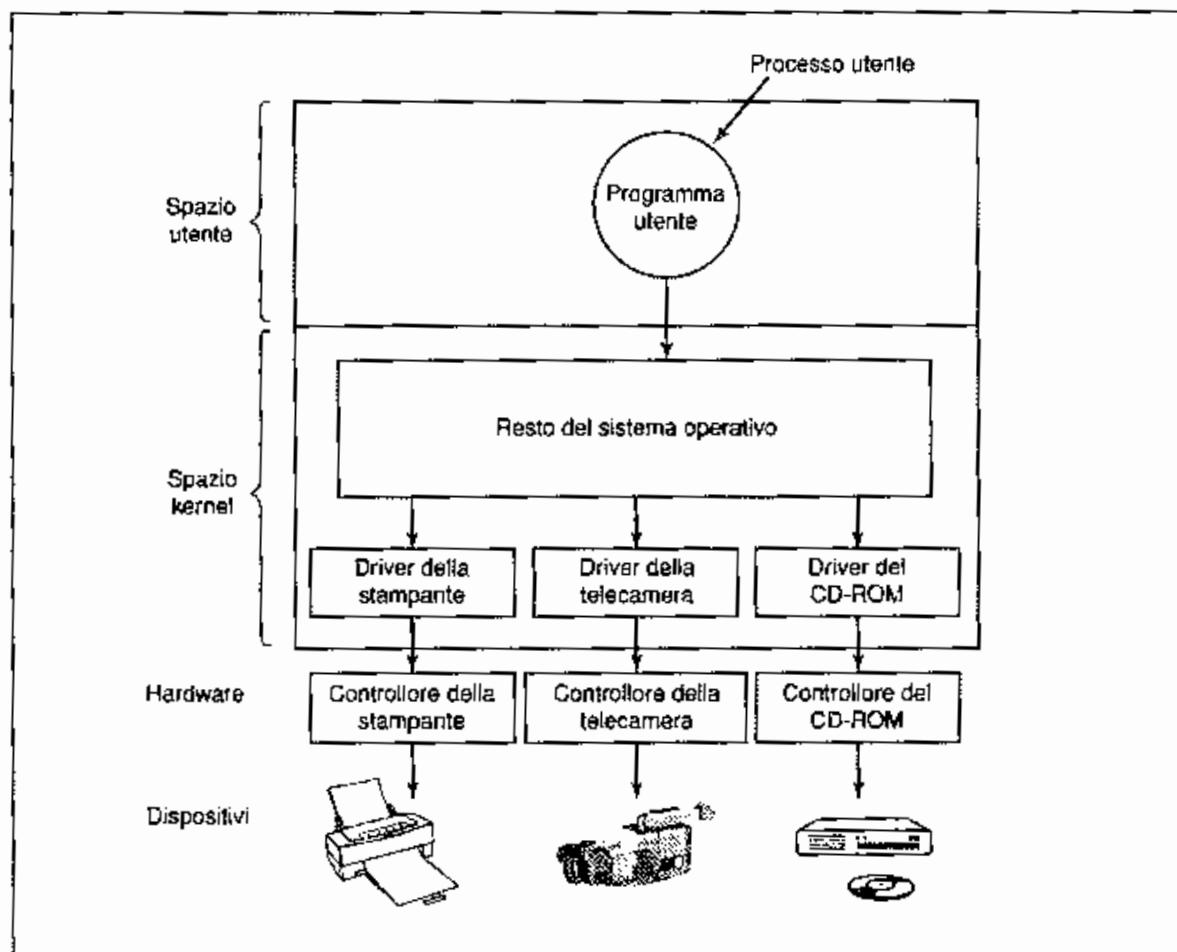


Figura 5.11 Posizionamento logico dei driver di dispositivo. In realtà tutta la comunicazione tra driver e controllori di dispositivo avviene attraverso il bus.

D'altra parte, un mouse e un joystick sono così diversi che solitamente richiedono driver diversi. Tuttavia non c'è nessuna restrizione tecnica che impedisca a un driver di dispositivo di controllare più dispositivi di diverso tipo, ma semplicemente, non si tratta di una buona idea.

Allo scopo di accedere all'hardware di un dispositivo, ossia ai registri del controllore, un driver deve essere parte del kernel del sistema operativo, quanto meno per le architetture moderne; in effetti sarebbe possibile costruire dei driver che possano essere eseguiti nello spazio utente, con chiamate di sistema che leggano e scrivano i registri del dispositivo e anzi, questo tipo di progettazione potrebbe essere una buona idea, nel senso che isolerebbe il kernel dai driver, e i driver l'uno dall'altro. Una scelta di questo tipo eliminerebbe una delle fonti principali dei crash di sistema: driver che contengono errori, i quali interferiscono con il kernel in un modo o nell'altro. Comunque, i sistemi operativi moderni si aspettano che i driver vengano eseguiti nello spazio kernel, e quindi nel seguito verrà considerata questa scelta.

Siccome i progettisti di sistemi operativi saranno che dei pezzi di codice (i driver) scritti da terzi saranno installati all'interno del sistema operativo, esso dovrà avere un'architettura che permetta queste installazioni; ossia, questo significa avere un modello ben definito di quello che un driver fa, e di come interagisce con il resto del sistema operativo. I driver di dispositivo sono normalmente posizionati al di sotto del resto del sistema operativo, come illustrato in Figura 5.11.

I sistemi operativi solitamente classificano i driver in poche categorie. Tra le più comuni ci sono i **dispositivi a blocco**, quali i dischi, che contengono molti blocchi di dati ai quali si può accedere in modo indipendente, e i **dispositivi a carattere**, quali le tastiere e le stampanti, che generano o accettano un flusso di caratteri.

La maggior parte dei sistemi operativi definisce un'interfaccia standard che tutti i dispositivi a blocchi devono supportare, e una seconda interfaccia standard che tutti i dispositivi a carattere devono supportare. Queste interfacce sono composte da un certo numero di procedure, che il resto del sistema operativo può chiamare per interagire con il driver. Procedure tipiche sono quelle che leggono un blocco (per i dispositivi a blocchi) e quelle che scrivono una stringa di caratteri (per i dispositivi a carattere).

In alcuni sistemi, il sistema operativo è un unico programma binario, che contiene tutti i driver di cui potrebbe avere bisogno compilati al suo interno; questo schema è stato applicato per anni dai sistemi UNIX, in quanto giravano nei centri di calcolo, dove i dispositivi di I/O venivano cambiati raramente; nel caso si volesse aggiungere un nuovo dispositivo, l'amministratore di sistema ricompilava il kernel dopo avere inserito al suo interno il nuovo driver.

Con l'avvento dei personal computer e la loro miriade di dispositivi di I/O, questo modello non funzionava più: pochi utenti erano in grado di ricompilare il kernel, anche a partire dal sorgente e dai moduli oggetto, cosa non sempre possibile. Invece i sistemi operativi, a partire da MS-DOS, hanno scelto un modello in cui i driver vengono caricati dinamicamente nel sistema durante l'esecuzione. Sistemi diversi carichano i driver in modi diversi.

Un driver di dispositivo ha diverse funzioni, la più ovvia delle quali è quella di accettare richieste di lettura e di scrittura astratte, da parte del software indipendente dal dispositivo che si trova al livello più alto, e verificare che vengano eseguite; ci sono però altre funzioni; per esempio, il driver deve inizializzare il dispositivo se necessario, gestire l'alimentazione, e tenere un log degli eventi.

Molti driver di dispositivo hanno la stessa struttura generale; il driver tipico comincia controllando i parametri di input e verificandone la validità; nel caso non siano validi ritorna con un messaggio di errore. Se sono validi, può essere necessaria una traduzione tra termini astratti e termini concreti; per un driver di disco questo può significare che il

numero del blocco lineare deve essere convertito in numeri di testina, traccia, settore e cilindro per rispettare la geometria del disco.

In seguito il driver deve controllare se il dispositivo è in uso e, nel caso lo sia, le eventuali richieste devono essere messe in coda per un'elaborazione successiva; se il dispositivo è inattivo, si esaminerà lo stato dell'hardware, per controllare che la richiesta possa essere eseguita. In questo caso, può essere necessario accendere il dispositivo o far partire un motore prima che il trasferimento possa iniziare. Quando il dispositivo è pronto a funzionare ha inizio il controllo vero e proprio.

Controllare un dispositivo significa emettere una sequenza di comandi verso di esso: il driver è il luogo in cui si decide la sequenza di comandi, a seconda di quello che c'è da fare. Dopo che il driver ha deciso quali comandi emettere, inizia a scriverli nei registri del controllore del dispositivo; dopo avere scritto un comando, può essere necessario verificare se il controllore lo ha accettato ed è pronto a ricevere il successivo. Questa sequenza continua finché tutti i comandi non sono stati emessi; alcuni controllori accettano liste concatenate di comandi in memoria, e le leggono ed eseguono in modo autonomo senza ricevere aiuto dal sistema operativo.

Dopo che i comandi sono stati emessi, si verifica una delle due situazioni seguenti: in molti casi il driver di dispositivo deve aspettare qualche azione da parte del controllore, quindi si blocca finché una qualche interruzione non lo sblocca; in altri casi, comunque, l'operazione viene terminata senza ritardi, quindi il driver non viene bloccato. Come esempio di quest'ultima situazione, si pensi che scorrere lo schermo in modo carattere richiede semplicemente la scrittura di pochi byte nei registri del controllore, senza compiere nessun movimento meccanico, quindi tutta l'operazione può essere completata in pochi nanosecondi.

Nel primo caso, il driver bloccato verrà svegliato da un'interruzione; nel secondo, il driver non avrà bisogno di essere svegliato perché non era mai andato a dormire. In entrambi i casi, dopo che l'operazione è stata completata, il driver deve effettuare un controllo sugli eventuali errori; se tutto è andato bene probabilmente il driver avrà dei dati da passare al software indipendente dal dispositivo (ad esempio, il blocco appena letto). Infine, il driver ritorna al suo chiamante qualche informazione di stato sulla situazione degli errori. Nel caso ci fossero altre richieste in coda, ne viene scelta un'altra e si ricomincia da capo, altrimenti i driver si blocca in attesa della richiesta successiva.

Questo modello semplice è solo una rozza approssimazione della realtà; molti fattori rendono il codice molto più complicato. Prima di tutto, un dispositivo di I/O può completare un'operazione mentre il driver è occupato, interrompendolo. Tale interruzione può provocare l'esecuzione di un driver e, in effetti, può fare sì che il driver corrente venga eseguito; ad esempio, mentre il driver di rete sta elaborando un pacchetto in ingresso, può arrivare un altro pacchetto. Di conseguenza, i driver devono essere rientranti, ossia un driver deve aspettarsi di poter essere chiamato di nuovo prima che la chiamata corrente sia stata completata.

In un sistema collegabile "a caldo", si possono aggiungere o rimuovere dispositivi, mentre il computer è in esecuzione; il risultato è che, mentre un driver è occupato a leggere da un dispositivo, può succedere che il sistema lo informi che un utente ha improvvisamente deciso di rimuovere quel dispositivo dal sistema. Non solo occorre abortire il trasferimento di I/O corrente, senza danneggiare le strutture dati del kernel, ma bisogna rimuovere dal sistema ogni richiesta pendente verso il dispositivo appena scomparso, ed occorre anche dare a tutti i richiedenti la brutta notizia. Inoltre, un'aggiunta inattesa di nuovi dispositivi fa sì che il kernel debba riadattare le risorse (ad esempio linee di richiesta di interruzioni), togliendo quelle vecchie dai driver e asseguando delle nuove ai loro posto.

I driver non possono effettuare chiamate di sistema, ma spesso devono interagire col

resto del kernel. Solitamente sono permesse chiamate verso alcune procedure del kernel; ad esempio, solitamente ci sono chiamate che allocano e deallocano pagine di memoria per usarle come buffer. Altre chiamate utili sono necessarie per la gestione della MMU, dei timer, del controllore di DMA, del controllore delle interruzioni e così via.

5.3.3 Software di I/O indipendente dai dispositivi

Sebbene parte del software di I/O sia specifico di un qualche dispositivo, esso è in larga misura indipendente dai dispositivi fisici. Il confine esatto fra i driver e il software indipendente dai dispositivi dipende dal sistema (e dal dispositivo), perché alcune funzioni che potrebbero essere svolte dalla parte indipendente dai dispositivi, possono in realtà essere svolte dai driver, per ragioni di efficienza o per altre ragioni. Le funzioni mostrate in Figura 5.12, sono tipicamente svolte dal software indipendente dai dispositivi.

La funzione principale del software indipendente dai dispositivi, è di eseguire le operazioni di I/O che sono comuni a tutti i dispositivi, e di fornire un'interfaccia uniforme al software di livello utente. Nel seguito analizzeremo i concetti descritti qui sopra con maggiore dettaglio.

Interfacciamento uniforme per i driver di dispositivo

Un punto importante, in un sistema operativo, è cercare di rendere tutti i dispositivi e i driver di I/O più simili possibile. Se dischi, stampanti, tastiere, eccetera, sono interfacciati in modi diversi, ogni volta che arriva un nuovo dispositivo, il sistema operativo deve essere modificato, e questa manipolazione non sarebbe una buona idea.

Un aspetto di questo problema è l'interfaccia tra driver di dispositivo e il resto del sistema operativo. In Figura 5.13 (a) illustriamo una situazione in cui ogni driver di dispositivo ha un'interfaccia diversa col sistema operativo, questo significa che le funzioni del driver che il sistema può chiamare differiscono da driver a driver; può anche significare che le funzioni del kernel di cui il driver ha bisogno variano da driver a driver. Mettendo insieme tutti questi problemi, si giunge alla conclusione che in questo modo interfacciare ogni nuovo driver richiede un grande sforzo di programmazione.

In contrasto, nella Figura 5.13(b) mostriamo un diverso approccio in cui tutti i driver hanno la stessa interfaccia; in questo modo diventa molto più semplice inserire un nuovo driver, supponendo che questo sia conforme all'interfaccia comune. Inoltre significa che i programmatore di driver sanno bene quello che viene loro richiesto (per esempio, quali funzioni debbano fornire e quali funzioni del kernel possono chiamare). In pratica, non tutti i dispositivi sono perfettamente identici, ma solitamente esiste solo un numero limitato di tipi di dispositivi e anche questi sono solitamente molto simili tra loro. Per esempio, persino dispositivi a blocchi e dispositivi a caratteri hanno molte funzioni in comune.

Interfaccia uniforme per driver di dispositivo
Bufferizzazione
Segnalazione degli errori
Allocazione e rilascio di dispositivi dedicati
Fornire una dimensione del blocco indipendente dal dispositivo

Figura 5.12 Funzioni del software indipendente dai dispositivi.

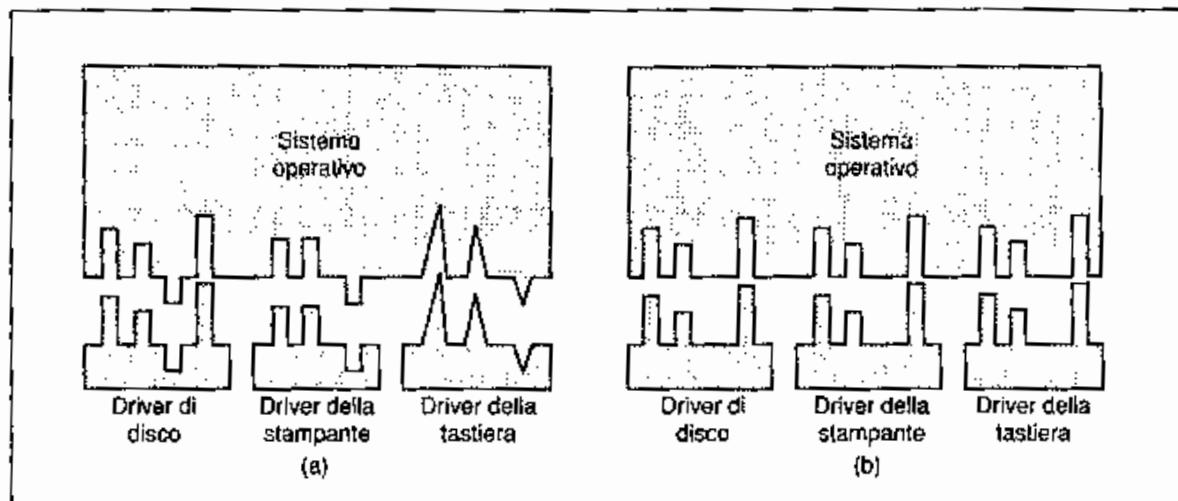


Figura 5.13 [a] Senza un'interfaccia di driver standard. [b] Con un'interfaccia di driver standard.

Un altro aspetto di avere un'interfaccia comune è il modo in cui sono denominati i dispositivi di I/O. Il software indipendente dai dispositivi si prende cura di tradurre i nomi simbolici delle periferiche con i nomi dei driver opportuni. In UNIX, per esempio, un nome di dispositivo, come `/dev/disk0`, definisce esattamente l'i-node di un file speciale che contiene il **numero principale di dispositivo** (major device number) usato per trovare il driver appropriato. L'i-node contiene anche il **numero di dispositivo secondario** (minor device number) che viene passato come parametro al driver, per specificare l'unità su cui si deve leggere o scrivere. Tutti i dispositivi hanno un numero principale e un numero secondario, e si accede a tutti i driver usando il numero principale del dispositivo per selezionare il driver.

Il problema della protezione è strettamente correlato al problema della denominazione degli oggetti. In che modo il sistema operativo evita che un utente acceda ad un dispositivo al quale non ha diritto di accedere? Sia in UNIX sia in Windows 2000 i dispositivi appaiono all'interno del file system come oggetti con nome, il che significa che le solite regole di protezione dei file si applicano anche ai dispositivi di I/O, e l'amministratore del sistema può selezionare le protezioni opportune per ogni dispositivo.

Bufferizzazione

Per una serie di ragioni, un altro concetto importante da trattare è quello di bufferizzazione, sia per dispositivi a blocchi sia per dispositivi a caratteri. Analizziamo una di queste ragioni considerando un processo che voglia leggere dati da un modem. Una strategia possibile di gestire i dati in entrata è quella di avere un processo utente che fa una chiamata di sistema `read` e si blocca in attesa di un carattere; ognuno dei caratteri in arrivo causa un'interruzione, e la procedura di servizio delle interruzioni consegna il carattere al processo utente e lo sblocca; il processo mette il carattere da qualche parte, legge un altro carattere e si blocca nuovamente. Questo modello è descritto in Figura 5.14(a).

Il problema di questo schema di azione è che il processo utente deve essere fatto ripartire per ogni carattere in arrivo; permettere ad un processo di andare in esecuzione molte volte per tempi brevi è molto inefficiente, quindi questo sistema non va bene.

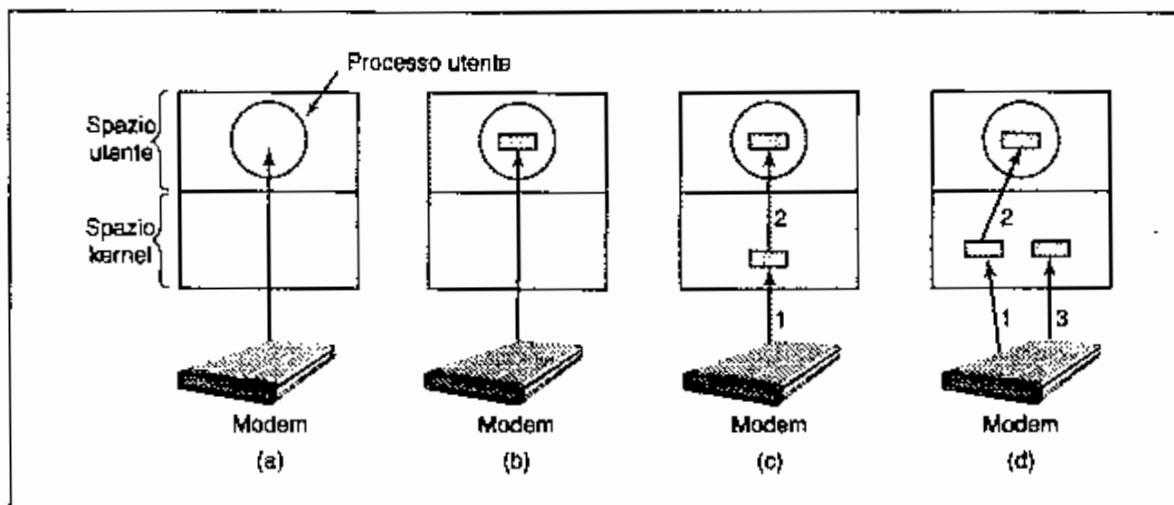


Figura 5.14 (a) Input non bufferizzato. (b) Bufferizzazione nello spazio utente. (c) Bufferizzazione nel kernel seguita da copia nello spazio utente. (d) Doppia bufferizzazione nel kernel.

Uno schema migliore è mostrato in Figura 5.14(b). In questo caso il processo utente fornisce un buffer di n caratteri, e la procedura di servizio delle interruzioni mette i caratteri in ingresso in questo buffer finché questo non si riempie; a questo punto sveglia il processo utente. Questo schema è molto più efficiente del precedente, ma anch'esso ha i suoi lati negativi: cosa succede se il buffer viene scaricato dalla paginazione della memoria mentre sta arrivando un nuovo carattere? Il buffer potrebbe venir bloccato in memoria, ma se molti processi iniziano a bloccare pagine in memoria, le pagine disponibili diminuiranno e le prestazioni ne risentiranno.

Un altro approccio consiste nel creare un buffer all'interno del kernel, e fare in modo che il driver delle interruzioni metta i caratteri in questo buffer, come mostrato in Figura 5.14(c). Quando questo buffer è pieno, si carica la pagina con il buffer utente, se necessario, e si copia il buffer in una singola operazione. Questo schema è molto più efficiente.

Comunque anche questo schema ha dei problemi: cosa succede ai caratteri che arrivano mentre la pagina contenente il buffer utente viene portata in memoria dal disco? Siccome il buffer nel kernel è pieno, non vi è posto dove metterle. Una soluzione è quella di avere un altro buffer nel kernel, come mostrato in Figura 5.14(d). Quando il primo buffer si riempie, diventa possibile copiarlo nello spazio utente (supponendo che l'utente lo abbia richiesto) e, mentre il primo buffer viene copiato nello spazio utente, si può usare il secondo per i nuovi caratteri. In questo modo i due buffer fanno a turno: mentre se ne copia uno nello spazio utente, l'altro può accumulare il nuovo input. Questo schema di bufferizzazione è chiamato **bufferizzazione doppia**.

La bufferizzazione è un'operazione importante anche sull'output. Si consideri per esempio come si può gestire l'output verso il modem senza uno schema di bufferizzazione come quello descritto in Figura 5.14(b): il processo utente esegue una chiamata di sistema `write` per fare l'output di n caratteri; a questo punto il sistema ha due scelte: può bloccare l'utente finché non sono stati scritti tutti i caratteri, ma questo potrebbe richiedere molto tempo su una linea telefonica lenta, oppure può liberare immediatamente l'utente e procedere con l'operazione di I/O, mentre l'utente prepara altri dati da mettere in output. Questo porta ad un problema anche peggiore: come fa il processo utente a sapere che l'output è stato completato e che può riusare il buffer? Il sistema potrebbe generare un segnale o un'interruzione software, ma questo stile di program-

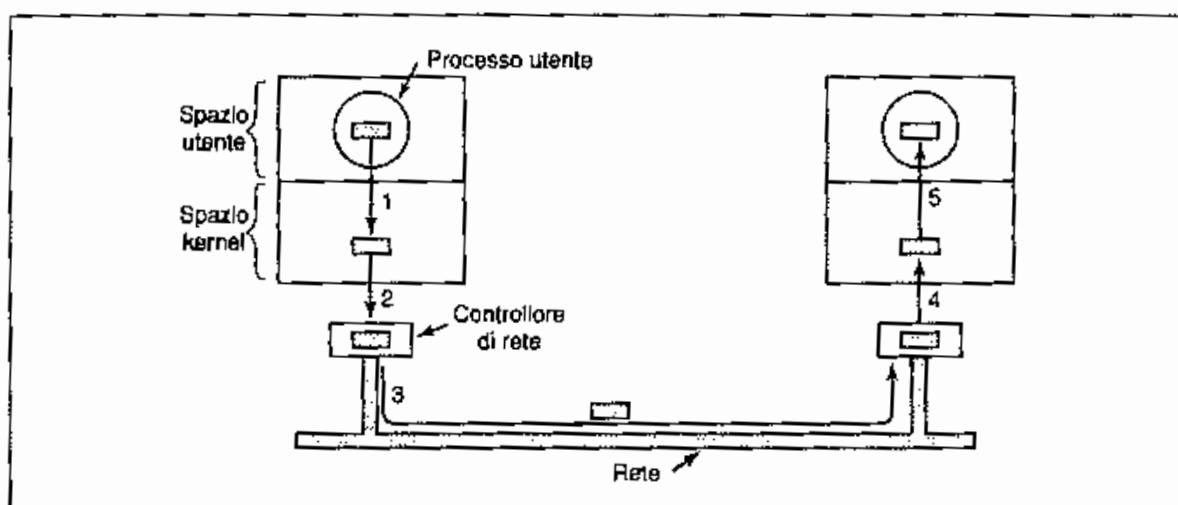


Figura 5.15 L'uso della rete può richiedere molte copie di uno stesso pacchetto.

mazione è difficile, e soggetto a condizioni di corsa critica. Una soluzione molto migliore è quella di ricopiare i dati in un buffer del kernel, analogamente a quanto accade in Figura 5.14(c) (ma nell'altra direzione), e sbloccare il chiamante immediatamente. A questo punto non importa che l'I/O sia stato completato, e l'utente è libero di riusare il buffer dall'istante in cui viene sbloccato.

La bufferizzazione è una tecnica ampiamente usata, ma ha i suoi lati negativi: se viene usata troppo spesso, le prestazioni ne risentono. Consideriamo ad esempio la rete in Figura 5.15, dove un utente fa una chiamata di sistema per scrivere sulla rete: il kernel copia il pacchetto in un buffer del kernel e lascia l'utente procedere immediatamente (passo 1).

Quando il driver viene chiamato, copia il pacchetto nel controllore per metterlo in output (passo 2); la ragione per cui non lo fa uscire direttamente dalla memoria del kernel è che, nel momento in cui si inizia a trasmettere un pacchetto, si deve continuare ad una velocità uniforme, ed il driver non può garantire di accedere alla memoria a velocità uniforme in quanto i canali del DMA e di altri dispositivi I/O possono tenere occupati molti cicli. Il pacchetto potrebbe venire rovinato se manca la temporizzazione nella trasmissione di una parola, mentre il problema può essere risolto mettendo l'intero pacchetto all'interno del buffer del controllore.

Dopo che il pacchetto è stato copiato nel buffer interno del controllore, esso viene copiato sulla rete (passo 3). I bit arrivano al destinatario subito dopo essere stati spediti, quindi subito dopo che l'ultimo bit è stato spedito, tali bit iniziano ad arrivare al destinatario, dove il pacchetto viene bufferizzato all'interno del controllore. Successivamente il pacchetto viene copiato nel buffer del kernel del destinatario (passo 4) ed infine copiato nel buffer del processo ricevente (passo 5). Solitamente il destinatario spedisce un acknowledgement, e quando il mittente lo riceve sa di essere libero di spedire il pacchetto successivo. Dovrebbe comunque essere chiaro che tutto questo copiare finisce per rallentare notevolmente il tasso di trasferimento, dato che tutti i passi appena descritti devono accadere in modo sequenziale.

Riportare errori

Gli errori sono molto più comuni nel contesto dell'I/O di quanto non lo siano in altri contesti; quando si verificano, il sistema operativo deve gestirli nel miglior modo possi-

bile. Molti errori sono specifici di determinati dispositivi, e devono quindi essere gestiti dal driver apposito, ma la gestione degli errori avviene in un contesto indipendente dal dispositivo.

Una classe di errori di I/O sono errori di programmazione che si verificano quando un processo chiede qualcosa di impossibile, come ad esempio scrivere su un dispositivo di input (tastiera, mouse, scanner, eccetera) o leggere da un dispositivo di output (stampante, plotter, eccetera). Altri errori sono: fornire indirizzi di buffer o altri parametri non validi, e specificare un dispositivo non valido (ad esempio, il disco numero 3 in un sistema con solo due dischi). Le azioni da intraprendere in questi casi sono dirette: semplicemente, si riporta un codice di errore al chiamante.

Un'altra classe di errori è la classe di errori di I/O veri e propri, per esempio cercare di scrivere su un blocco di disco che è rovinato, o cercare di leggere da una telecamera che è spenta. In queste circostanze il driver deve decidere cosa fare; se non sa cosa fare, può passare il problema all'indietro al software indipendente dai dispositivi.

Quello che questo software decide di fare dipende dall'ambiente e dalla natura dell'errore; se è un semplice errore di lettura, e c'è un utente interattivo disponibile, può visualizzare una finestra di dialogo chiedendo all'utente cosa fare. Le opzioni possono essere: ritentare un certo numero di volte, ignorare l'errore, o uccidere il processo. Nel caso non vi sia un utente, probabilmente l'unica cosa da fare è di fare fallire la chiamata di sistema, e restituire un codice di errore.

Comunque, alcuni errori non possono essere gestiti in questo modo, per esempio se una struttura dati critica, quale la directory radice o la lista dei blocchi liberi, viene distrutta, in questo caso il sistema può dover mostrare un messaggio di errore e terminare.

Allocare e rilasciare dispositivi dedicati

Alcuni dispositivi, quali i masterizzatori di CD-ROM, possono essere usati solo da un singolo processo alla volta. Il sistema operativo ha il compito di esaminare le richieste e accettarle o rifiutarle, a seconda che il dispositivo richiesto sia disponibile o meno. Un modo semplice per gestire queste richieste è quello di chiedere al processo di effettuare direttamente delle open sui file speciali dei dispositivi, in quanto se il dispositivo è occupato la open fallirà, mentre una close del file speciale rilascerà il dispositivo.

Un approccio alternativo è quello di avere meccanismi speciali che richiedano e rilascino i dispositivi dedicati; un tentativo di acquisire un determinato dispositivo che non sia disponibile blocca il chiamante invece di fallire. I processi bloccati vengono posti in una coda e prima o poi, quando il dispositivo verrà liberato, sarà assegnato al primo processo nella coda, il quale lo potrà acquisire e continuare l'esecuzione.

Dimensione dei blocchi indipendente dal dispositivo

Dischi diversi possono avere settori di diverse dimensioni; quindi sta al software indipendente dal dispositivo nascondere questo fatto, e fornire una dimensione dei blocchi uniforme ai livelli più alti, ad esempio trattando molti settori come un unico blocco logico. In questo modo, i livelli più alti hanno solo a che fare con dispositivi astratti che usano la stessa dimensione logica, indipendentemente dalla dimensione fisica del settore. In modo simile, alcuni dispositivi a carattere, consegnano il loro dati un byte alla volta (ad esempio i modem), mentre altri consegnano i dati in unità più grandi (ad esempio le interfacce di rete); anche queste differenze possono essere nascoste.

5.3.4 Software di I/O del livello utente

Sebbene la maggior parte del software di I/O si trovi all'interno del sistema operativo, una piccola parte di esso si compone di librerie collegate ai programmi utente, e da interi programmi che risiedono completamente al di fuori del nucleo del sistema operativo. Le chiamate di sistema, comprese le chiamate relative all'I/O, sono normalmente realizzate tramite procedure di libreria. Quando un programma C contiene la chiamata

```
cont = write(fd, buffer, nbytes);
```

la procedura di libreria `write` sarà collegata al programma, e sarà presente nel programma binario caricato in memoria al momento dell'esecuzione. L'insieme di tutte queste librerie fa chiaramente parte del sistema di I/O.

Nonostante queste procedure facciano poco di più che passare dei parametri al posto giusto alla chiamata di sistema, ci sono altre procedure relative all'I/O che svolgono realmente del lavoro; in particolare, la formattazione dei dati di ingresso e uscita è svolta da procedure di libreria. Un esempio, tratto dal C, è la funzione `printf`, che prende in ingresso una stringa di formato e alcune variabili, costruisce una stringa di caratteri ASCII e chiama la procedura `write` per stampare la stringa. Come esempio di `printf`, consideriamo il seguente comando:

```
printf("Il quadrato di %3d è %6d\n", i, i*i);
```

Esso formatta una stringa consistente nella stringa di 15 caratteri "Il quadrato di " seguita dal valore `i` come stringa di 3 caratteri, e poi dalla stringa di 3 caratteri " è ", poi dai 6 caratteri di i^2 ed infine il carattere di fine linea.

Un esempio simile, ma relativo all'input, è la funzione `scanf`, che prende i caratteri in ingresso e immagazzina i valori nelle variabili, descritte da una stringa di formattazione usando la stessa sintassi della `printf`. La libreria standard di I/O contiene un gran numero di funzioni simili, che riguardano l'I/O, e tutte girano come parte dei programmi utente.

Non tutto il software di I/O di livello utente è costituito da procedure di libreria. Una categoria importante è il sistema di spool (funzionamento simultaneo delle periferiche). Lo spooling è una maniera particolare di trattare i dispositivi dedicati in un ambiente multiprogrammato. Si consideri, per esempio, un tipico dispositivo su cui di solito si fa lo spooling: la stampante; sebbene sia tecnicamente possibile lasciare che un processo utente apra il file speciale che corrisponde alla stampante, si supponga che un processo apra tale file e poi non lo adoperi per molto tempo senza chiuderlo: in questo caso nessun altro processo potrebbe usare la stampante.

Quello che si fa di solito, allora, è creare un processo speciale, chiamato demone, e una directory speciale, chiamata directory di spool. Per stampare un file, un processo prima genera tutto il file da stampare, e poi lo scrive nella directory di spool. In seguito il demone, che è l'unico processo ad avere i diritti sul file speciale che rappresenta la stampante, stampa i file presenti nella directory di spool. In questa maniera, proteggendo il file speciale dall'uso diretto da parte degli utenti, si risolve il problema di avere la stampante occupata per un tempo più lungo del necessario.

La tecnica dello spooling non viene usata solamente per le stampanti, ma anche in altre situazioni. Per esempio, il trasferimento dei file su una rete viene spesso eseguito con l'aiuto di un demone; infatti per spedire un file da qualche parte, l'utente non fa altro che scriverlo nella directory di spool di rete. Più tardi, il demone lo prenderà dalla directory e lo spedirà effettivamente sulla rete. Un uso particolare della trasmissione di file con lo spooling è fatto dal sistema di News USENET; la rete è costituita da milioni di mac-

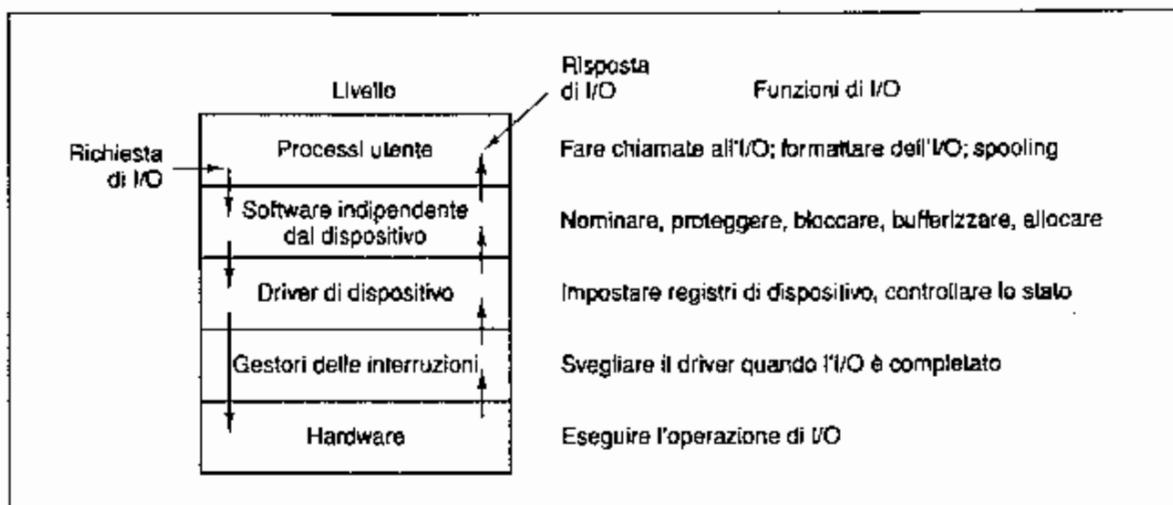


Figura 5.16 I livelli di un sistema di I/O e le funzioni principali di ogni livello.

chine sparse nel mondo che comunicano fra di loro mediante Internet. Esistono migliaia di *news group* (gruppi di notizie) dedicati a molti argomenti; per mandare un messaggio di news, l'utente chiama il programma delle news, il quale accetta il messaggio da spedire e lo deposita in una directory di spooling dalla quale verrà successivamente spedito ad altre macchine. L'intero sistema delle news viene eseguito al di fuori del sistema operativo.

La Fig. 5.16 riassume il sistema di I/O, e mostra tutti i suoi livelli e le loro principali funzionalità. Cominciando dal basso, i livelli sono l'hardware, i gestori delle interruzioni, i driver di dispositivo, il software indipendente dal dispositivo, ed infine i processi utente.

Le frecce nella Fig. 5.16 mostrano il flusso di controllo. Quando un programma utente, per esempio, prova a leggere un blocco da un file, viene chiamato il sistema operativo per gestire la chiamata. Il software indipendente dalla periferica cerca nella cache dei buffer: se il blocco necessario non è presente, chiama il driver della periferica per passare la richiesta all'hardware del disco. Il processo è bloccato finché non è completata l'operazione del disco.

Quando il disco ha terminato, l'hardware genera un'interruzione. Il driver delle interruzioni va alla ricerca di quello che è accaduto, cioè cerca quale è la periferica che ha richiamato la sua attenzione; poi prende lo stato della periferica, ed attiva il processo bloccato affinché termini la sua richiesta di I/O, ed il processo utente possa continuare.

5.4 I dischi

Adesso inizieremo lo studio di alcuni dispositivi di I/O reali; inizieremo dai dischi, in seguito esamineremo i clock, le tastiere e i monitor.

5.4.1 L'hardware dei dischi

Sul mercato è disponibile una varietà di dischi; i più comuni sono i dischi magnetici (dischi fissi e dischetti), che sono caratterizzati dal fatto che la lettura e la scrittura sono ugualmente veloci, la qual cosa li rende molto adatti come memoria secondaria (paginazione, file system, eccetera). Talvolta si usano array di questi dischi per fornire meccani-

smi di memorizzazione altamente affidabili. Per la distribuzione di programmi, dati, film, vengono utilizzati vari tipi di dischi ottici (CD-ROM, CD-riscrivibili, DVD). Nelle pagine seguenti inizieremo col descrivere prima l'hardware e poi il software di tali dispositivi.

Dischi magnetici

Tutti i dischi fisici sono organizzati in cilindri, ognuno dei quali contiene tante tracce quante sono le testine sistematiche in verticale. Le tracce sono divise in settori, ed il numero dei settori che si trovano su una traccia va, generalmente, da 8 a 32 sui dischetti e fino a diverse centinaia sui dischi fissi. Il numero di testine, invece, varia all'incirca tra 1 e 16.

Alcuni dischi magnetici hanno una piccola parte di elettronica, quindi forniscono semplicemente flussi di bit seriali; su questi dischi il controllore fa la maggior parte del lavoro. Su altri dischi, in particolare gli IDE (Integrated Drive Electronics, elettronica integrata nell'attuatore), il drive stesso contiene un microcontrollore che fa parte del lavoro, e lascia al controllore vero e proprio il compito di emettere comandi a livello più alto.

Un'importante caratteristica del dispositivo, che ha conseguenze importanti per il driver dei dischi, è la possibilità che un controllore esegua operazioni di posizionamento (seek) su due o più dischi contemporaneamente. Questo fatto è noto come **posizionamenti sovrapposti** (overlapped seeks): mentre il controllore e il software stanno aspettando il completamento di un'operazione di posizionamento su un disco, il controllore può cominciare un'operazione di posizionamento su un altro disco. Molti controllori possono anche leggere o scrivere su di un disco mentre stanno effettuando operazioni di posizionamento su uno o più dischi diversi, ma nessun controllore di floppy disk è in grado di leggere o scrivere contemporaneamente da due dischi. (Leggere o scrivere richiede che il controllore sposti dei bit in un tempo dell'ordine dei microsecondi, per cui un singolo trasferimento usa la maggior parte della sua potenza di calcolo). La situazione è diversa per dischi fissi con controllori integrati, e in sistemi con più dischi fissi essi possono operare simultaneamente, almeno fino al punto di effettuare trasferimenti tra dischi e il buffer di memoria del controllore; comunque, è possibile solo un trasferimen-

Parametri	Floppy disk IBM 360-KB	Disco fisso WD 18300
Numero di cilindri	40	10601
Tracce per cilindro	2	12
Settori per tracce	9	281 (media)
Settori per disco	720	35742000
Byte per settore	512	512
Capacità del disco	360 KB	18,3 GB
Tempo di posizionamento (cilindri adiacenti)	6 ms	0,8 ms
Tempo di posizionamento (caso medio)	77 ms	6,9 ms
Tempo di rotazione	200 ms	8,33 ms
Tempo del motore per fermarsi/partire	250 ms	20 s
Tempo per trasferire un settore	22 ms	17 µs

Figura 5.17 Parametri dei dischi del PC IBM originale, con floppy disk a 360 KB, e del disco fisso Western Digital WD 18300.

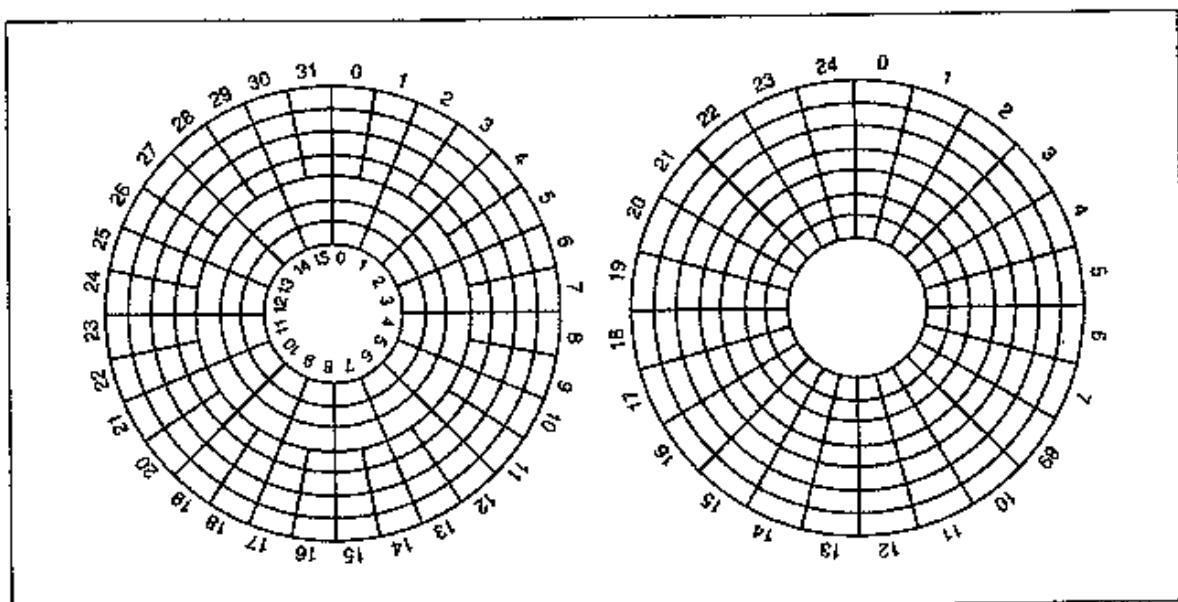


Figura 5.18 (a) Geometria fisica di un disco con due zone. (b) Una possibile geometria virtuale per lo stesso disco.

to alla volta tra controllore e memoria centrale. La possibilità di effettuare due o più operazioni allo stesso tempo può ridurre il tempo di accesso medio in modo sostanziale.

La Figura 5.17 confronta i parametri dei mezzi standard di memorizzazione per il PC IBM originale con i parametri di un moderno disco fisso, per mostrare quanto i dischi siano cambiati negli ultimi due decenni; è interessante notare che non tutti i parametri sono migliorati nello stesso modo. Il tempo medio di posizionamento è sette volte meglio, mentre il tempo di trasferimento è 1300 volte meglio, e la capacità è salita di un fattore di 50000. Questo andamento ha a che fare con il fatto che il miglioramento delle parti mobili è relativo, mentre la densità dei bit sulle superfici è decisamente aumentata.

Una cosa di cui essere consapevoli nell'analizzare le specifiche dei dischi fissi moderni è che la geometria specificata ed usata dal software del driver può essere diversa dalla formattazione fisica. Su dischi più vecchi, il numero di settori per traccia era lo stesso per tutti i cilindri, mentre i dischi moderni sono divisi in zone, con più settori in quelle esterne e meno in quelle interne. La Figura 5.18(a) mostra un piccolo disco con due zone: quella più esterna ha 32 settori per traccia, quella interna ha 16 settori per traccia. Un disco reale, ad esempio il WD 18300, spesso ha 16 zone, con il numero di settori che aumenta di circa il 4% per zona, se ci si muove dalla zona più interna a quella più esterna.

Per nascondere i dettagli di quanti settori risiedano in ogni traccia, la maggior parte dei dischi moderni è dotata di una geometria virtuale che viene presentata al sistema operativo. Il software è istruito a comportarsi come se ci fossero x cilindri, y testine e z settori per traccia; il controllore rimappa una richiesta per (x, y, z) sul vero cilindro, la vera testina e il vero settore. Una possibile geometria virtuale per il disco di Figura 5.18(a) è mostrata in Figura 5.18(b); in entrambi i casi il disco ha 192 settori, ma nel disco virtuale essi sono disposti diversamente rispetto al disco reale.

Per i computer basati sul processore Pentium, il valore massimo per questi tre valori è spesso (65535, 16 e 63), in quanto essi devono essere compatibili rispetto ai loro predecessori e quindi per le limitazioni imposte dal PC IBM originale. Su quella macchina venivano usati campi da 16, 4 e 6 bit per specificare tali numeri, e la numerazione di cilindri e settori iniziava da 1, mentre quella delle testine iniziava da 0. Con questi parametri,

e con 512 byte per settore, il disco più grande possibile è di 31,5 GB. Per aggirare questo limite, molti dischi supportano un sistema chiamato **indirizzamento a blocco logico**, in cui i settori sono numerati consecutivamente a partire dallo 0, senza considerare la geometria del disco.

RAID

Le prestazioni della CPU sono aumentate esponenzialmente nel corso dell'ultimo decennio, grosso modo raddoppiando ogni 18 mesi. Non è successo lo stesso alle prestazioni dei dischi: negli anni 70 il tempo di posizionamento medio dei dischi dei minicalcolatori era dai 50 ai 100 ms, mentre attualmente il tempo di posizionamento medio è poco al di sotto dei 10 ms. Nella maggior parte delle industrie tecniche (ad esempio automobilistiche o dell'aviazione), un fattore di miglioramento delle prestazioni, nell'arco di vent'anni, di 5 o 10 sarebbe una buona notizia, ma nell'industria del computer questo è fonte di imbarazzo. Così la differenza tra le prestazioni della CPU e del disco è aumentata notevolmente nel corso degli anni.

Come abbiamo visto, l'elaborazione parallela viene usata sempre di più per aumentare le prestazioni della CPU; nel corso degli anni molti hanno pensato che anche la parallelizzazione dell'I/O potrebbe essere una buona idea. Patterson et al., nel loro articolo del 1988, suggerirono sei organizzazioni particolari del disco, che avrebbero potuto migliorare le prestazioni dello stesso e la sua affidabilità, o entrambe le cose (Patterson et al., 1988). Queste idee sono state velocemente adottate dall'industria, ed hanno portato ad una nuova classe di dispositivi di I/O chiamata **RAID**. Patterson et al. scelsero l'acronimo RAID da **array ridondante di dischi economici** (Redundant Array of Inexpensive Disks), ma l'industria ha ridefinito la I come "Independent" (indipendenti) invece che "Inexpensive" (economici) (forse perché in questo modo potevano usare dischi non economici?). Siccome c'era bisogno di un antagonista (come in RISC contro CISC, anche questa dovuta a Patterson), il cattivo in questo caso era lo **SLED** (*disco unico grande e costoso, Single Large Expensive Disk*).

L'idea principale di RAID è quella di installare una scatola piena di dischi accanto ad un computer, tipicamente un grande server, sostituire la scheda del controllore del disco con un controllore RAID, copiare i dati sul RAID e continuare le normali operazioni. In altre parole un RAID dovrebbe apparire come uno SLED al sistema operativo, ma avere migliori prestazioni e una maggiore affidabilità. Siccome i dischi SCSI hanno buone prestazioni, prezzo basso e la possibilità di avere fino a 7 drive su un unico controllore (15 per SCSI ampi), è naturale che la maggior parte dei RAID siano composti da un controllore RAID SCSI, più un insieme di dischi SCSI che appaiono al sistema operativo come un unico disco. In questo modo non si devono fare cambiamenti al software per usare un RAID, la qual cosa ne favorisce la popolarità tra gli amministratori di sistema.

In aggiunta al fatto di sembrare un unico disco al sistema operativo, i RAID hanno la proprietà di distribuire i dati su dischi diversi, favorendo l'elaborazione parallela. Patterson et al. definirono molti schemi diversi d'organizzazione dati sull'array di dischi; questi schemi oggi sono noti come RAID livello 0, fino a RAID livello 5; in aggiunta a questi ci sono altri livelli minori che non discuteremo. Il termine "livello" può essere fuorviante, perché in questo caso non c'è alcun'idea di gerarchia, ma semplicemente 6 idee di possibili organizzazioni diverse.

Il RAID di livello 0 è illustrato in Figura 5.19(a); il disco virtuale singolo simulato dal RAID viene immaginato come se fosse diviso in fette (strip) di k settori l'una, dove i settori dallo 0 al $k - 1$ sono la fetta 0, i settori dal k al $2k - 1$ sono la fetta 1 e così via; per $k = 1$ ogni fetta corrisponde ad un settore, se $k = 2$ ogni fetta corrisponde a due settori,

eccetera. La configurazione di RAID livello 0 scrive su fette consecutive in modo round-robin, come descritto in Figura 5.19(a) per un RAID con quattro dischi. La tecnica con cui i dati vengono distribuiti su più drive viene chiamata **striping**, ossia i dati sono sistemati a strisce sui dischi disponibili. Ad esempio, se il software emette un comando di lettura di un blocco di dati, che si compone di quattro fette consecutive, iniziando da un estremo della fetta, il controllore RAID spezzerà questo comando in quattro comandi separati, ognuno per un disco diverso, e li farà operare in parallelo: in questo modo si ha I/O parallelo senza che il software se ne accorga.

Il RAID di livello 0 lavora al meglio nel caso di grandi richieste, più grandi sono meglio è: se la richiesta è più grande del numero di drive moltiplicato per la dimensione di ogni fetta, alcuni drive riceveranno richieste multiple, in modo che quando completano la prima possono passare alla seconda. È compito del controllore dividere le richieste e assegnare ogni comando al disco giusto, nella sequenza giusta, ed in seguito riassumere i risultati in memoria in modo corretto. Le prestazioni sono eccellenti e l'implementazione è diretta.

Il RAID di livello 0 lavora peggio con i sistemi operativi che normalmente richiedono un settore alla volta: i risultati sono corretti, ma non viene effettuata nessuna parallelizzazione e quindi non si ha nessun guadagno nelle prestazioni. Un altro svantaggio di questa organizzazione è che l'affidabilità è potenzialmente peggiore rispetto all'usare uno SLED: se un RAID si compone di quattro dischi, ognuno dei quali con tempo medio tra un fallimento e l'altro di 20000 ore, circa una volta ogni 5000 ore uno dei quattro dischi fallirà, e tutti i dati saranno completamente persi; uno SLED con un tempo medio tra fallimenti di 20000 ore sarebbe quattro volte più affidabile. Siccome questo progetto non presenta alcun tipo di ridondanza, il RAID di livello 0 non è un RAID vero e proprio.

L'opzione successiva, il RAID di livello 1, mostrato in Figura 5.19(b), è un RAID vero e proprio. Esso duplica tutti i dischi, in modo che se ci sono quattro dischi primari, si hanno anche quattro dischi di backup. Durante un'operazione di scrittura, ogni fetta viene scritta due volte, mentre durante una lettura si può leggere una qualsiasi delle due copie, distribuendo il carico su drive diversi. Di conseguenza, le prestazioni della scrittura non sono migliori di quelle di un drive singolo, mentre quelle della lettura possono raddoppiare. La tolleranza ai guasti è eccellente: in caso di crash di un disco, al suo posto viene semplicemente usata la sua copia. Il ripristino da un fallimento non è altro che l'installazione di un nuovo drive, e il copiare su di esso il disco di backup.

A differenza dei livelli 0 e 1, che lavorano con fette formate da settori, il RAID di livello 2 lavora sulla base della parola o, se possibile, addirittura sulla base del byte. Si immagini di dividere ogni singolo byte del disco virtuale in due parti di 4 bit l'una, aggiungere un codice di Hamming ad ogni parte in modo da formare parole di 7 bit, delle quali i bit 1, 2 e 4 sono bit di parità; inoltre si immagini che i sette drive di Figura 5.19(c) siano sincronizzati in termini di posizione del braccio e posizione di rotazione. In questo modo sarebbe possibile scrivere la parola codificata di 7 bit sui sette drive, un bit per drive.

Il computer CM-2 della Thinking Machines usava questo schema, prendendo parole di 32 bit e aggiungendo 6 bit di parità in modo da formare parole di Hamming di 38 bit, più un altro bit di parità, e distribuendo ogni parola su 39 dischi. Il throughput totale era immenso, in quanto nel tempo di scrittura di un settore poteva scrivere 32 settori di dati. Inoltre, perdere un drive non avrebbe causato problemi, in quanto era equivalente a perdere un bit per ogni parola di 39 bit, cosa che la codifica di Hamming può gestire con tranquillità.

Il lato negativo di questo schema è che richiede che tutti i drive siano sincronizzati nel loro movimento rotatorio, e che ci sia un alto numero di dischi (anche con 32 drive dati e 6 drive di parità l'overhead è del 19 percento); inoltre richiede molto lavoro del controllore, in quanto deve fare un controllo di Hamming per ogni bit.

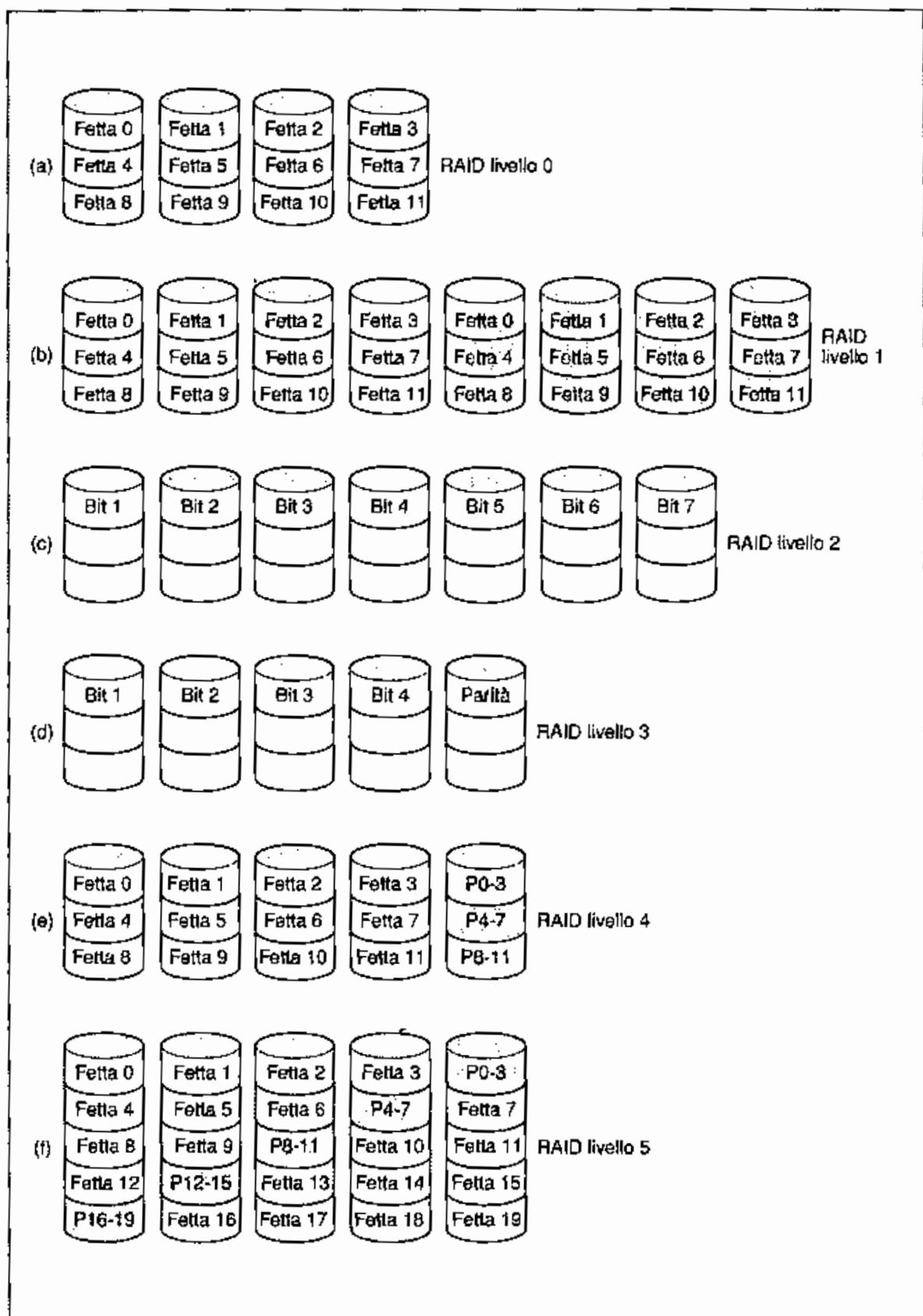


Figura 5.19 I livelli di RAID dallo 0 al 5. I drive di backup e di parità sono rappresentati in grigio.

Il RAID di livello 3 è la versione semplificata del RAID di livello 2. Esso è illustrato in Figura 5.19(d); in questo schema viene calcolato un solo bit di parità per ogni parola di dati, che viene scritto su un drive di parità. Come nel RAID livello 2, i drive devono essere sincronizzati in modo esatto, siccome ogni singola parola viene distribuita su più di un disco.

Potrebbe sembrare che un solo bit di parità possa fornire solo il rilevamento degli errori, ma non la correzione degli errori; nel caso di errori casuali non identificati questa osservazione è vera, mentre nel caso di crash del disco, esso fornisce una correzione dell'errore completa perché la posizione del bit corrotto è nota. Se si ha un crash del drive, il controllore fa finta che tutti i bit ivi presenti siano a 0; allora, se una parola ha un errore di parità, il bit relativo al drive guasto deve essere a 1, quindi l'errore può venire corretto. Anche se sia il RAID di livello 2 sia quello di livello 3 offrono frequenze di elaborazione dati molto alte, il numero di richieste di I/O distinte che possono servire al secondo non è superiore a quello di un drive singolo.

I RAID di livello 4 e 5 lavorano con fette, e non con singole parole più la parità, e non richiedono drive sincronizzati. Il RAID di livello 4 (vedi Figura 5.19(e)) è come RAID di livello 0, ma con un controllo di parità di ogni fetta scritto su un drive di parità. Se per esempio ogni fetta è lunga k byte, si effettua un OR ESCLUSIVO su di esse, ottenendo una fetta di parità di k byte. Se si ha un crash di disco, i byte persi possono essere ricostruiti dal drive di parità.

Questa organizzazione protegge contro la perdita di un disco, ma le sue prestazioni sono limitate nel caso di piccoli aggiornamenti. Se un settore è cambiato, è necessario rileggere tutti i drive per ricalcolare la parità e quindi riscriverla. Alternativamente, si possono leggere i vecchi dati e la vecchia parità, e calcolare quella nuova a partire da essi. Anche con questo miglioramento, un piccolo aggiornamento richiede comunque due letture e due scritture.

Come conseguenza del grosso carico che grava sul dispositivo di parità, esso può diventare un collo di bottiglia; questo problema è eliminato da RAID di livello 5, il quale distribuisce i bit di parità su tutti i drive, in modo round robin, come mostrato in Figura 5.19(f); comunque, in caso di crash di disco, ricostruire il contenuto del drive distrutto diventa un processo molto complesso.

CD-ROM

Negli ultimi anni si sono resi disponibili dischi ottici (in contrapposizione a quelli magnetici), i quali hanno densità di registrazione molto più alta dei tradizionali dischi magnetici. I dischi ottici furono sviluppati originariamente per la registrazione di programmi televisivi, ma hanno trovato un'applicazione più dignitosa come dispositivo di memorizzazione per calcolatori. Grazie alla loro capacità potenzialmente enorme, i dischi ottici sono stati oggetto di una grande quantità di ricerche, e si sono evoluti velocemente.

La prima generazione di dischi ottici fu inventata dalla sede olandese della Philips allo scopo di memorizzare film. Essi erano di 30 cm di diametro ed erano stati chiamati Laser-Vision, ma non presero piede tranne in Giappone.

Nel 1980 la Philips, insieme alla Sony, sviluppò il CD (Compact Disc), il quale sostituì rapidamente i dischi in vinile a 33 1/3 giri al minuto per la musica (tranne che tra i conoscitori, che continuano a preferire il vinile!). I dettagli tecnici precisi del CD furono pubblicati in uno standard internazionale ufficiale (IS 10149), noto come Libro Rosso, a causa del colore della sua copertina - gli standard internazionali sono emessi dalla Organizzazione Internazionale degli Standard (ISO, International Organization for Standardization) che è la controparte internazionale di gruppi nazionali per gli standard, quali ANSI,

DIN, eccetera. Ognuno ha un numero IS. Il motivo per voler specificare il disco e il drive come standard internazionali era di permettere la pubblicazione di CD da parte di diversi distributori musicali e di lettori CD da parte di diversi produttori di elettronica, tutti compatibili tra loro. Tutti i CD hanno un diametro di 120 mm e uno spessore di 1,2 mm, con un buco di 15 mm nel mezzo. Il CD audio è stato il primo mezzo di memorizzazione digitale ad avere uno sviluppo di massa. Essi dovrebbero durare 100 anni: per favore nel 2080 controllate se sono andati bene i primi CD prodotti.

Un CD viene preparato usando un laser a infrarossi ad alta intensità, che produce fori di 0,8 micron di diametro in un disco master di vetro ricoperto. Da questo master viene preparato un calco, con estrusioni al posto dei fori; all'interno di questo calco viene versata resina di policarbonato fusa, in modo da formare un CD con lo stesso pattern di fori del disco master. A questo punto un sottile strato di alluminio riflettente viene depositato sul policarbonato, ricoperto da una lacca protettiva, ed infine da un'etichetta. Le depressioni nel sottostato di policarbonato sono chiamate fosse (pit), mentre le aree non bruciate tra le fosse sono dette terre (land).

Quando un CD viene letto, un diodo laser di bassa intensità, emette una luce infrarossa con una lunghezza d'onda di 0,78 micron su fosse e terre; il laser è dal lato del policarbonato, quindi le fosse sono delle estrusioni in una superficie altrimenti piatta. Siccome hanno un'altezza di un quarto della lunghezza d'onda della luce laser, la luce riflessa da una fossa ha una lunghezza d'onda fuori fase della metà, rispetto a quella riflessa dalla superficie circostante. Il risultato è che le due luci (quella diretta e quella riflessa) interferiscono tra di loro in modo distruttivo, e restituiscono al fotorilevatore del lettore una quantità di luce minore di quella ricevuta quando il raggio laser colpisce una zona liscia: in questo modo il lettore distingue tra fossa e terra. Anche se può sembrare più semplice usare una fossa per registrare uno 0 e una terra per registrare un 1, è più affidabile usare una transizione da fossa a terra o viceversa per un 1, e l'assenza di transizione per uno 0, quindi è usato quest'ultimo schema.

Fosse e terre vengono scritte in un'unica spirale continua che inizia vicino al buco centrale e si dirige verso il bordo del disco, ad una distanza di circa 32 mm dal centro; la spi-

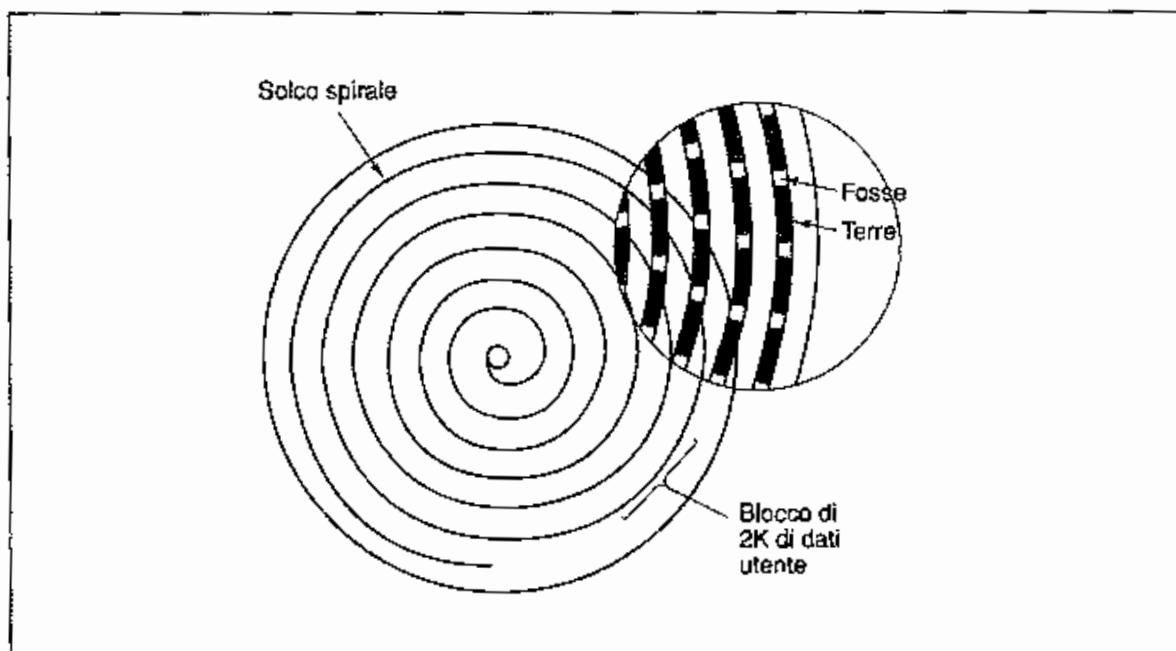


Figura 5.20 La struttura di registrazione di un compact disc o di un CD-ROM.

rale effettua circa 22188 rivoluzioni intorno al disco (circa 600 per mm); se srotolata sarebbe lunga 5,6 km. La spirale è illustrata in Figura 5.20.

Per fare in modo che la musica suoni ad una frequenza uniforme è necessario che fosse e terre si susseguano ad una velocità lineare costante, di conseguenza la frequenza di rotazione del CD deve essere continuamente ridotta man mano che la testina di lettura si muove verso l'esterno del CD. All'interno, la velocità di rotazione è di 530 giri al minuto, allo scopo di raggiungere la frequenza voluta di 120 cm/s; all'esterno deve scendere a 200 giri al minuto per ottenere la stessa velocità lineare sotto la testina. Un drive a velocità lineare costante è piuttosto diverso da un drive di disco magnetico, che opera a velocità angolare costante, indipendentemente dalla posizione della testina. È inoltre il caso di notare che 530 giri al secondo è abbastanza lontano dai 3600 o 7200 giri al secondo con cui si muovono la maggior parte dei dischi magnetici.

Nel 1984 la Philips e la Sony si accorsero della possibilità di memorizzare dati di elaboratori su CD e pubblicarono il Libro Giallo che definiva lo standard preciso di quelli che oggi sono noti come CD-ROM (CD con memoria a sola lettura, Compact Disk Read-Only Memory). Per adeguarsi al già fiorente mercato dei CD audio, i CD-ROM dovevano avere la stessa dimensione fisica, dovevano essere meccanicamente ed otticamente compatibili con essi, e prodotti usando le stesse apparecchiature a resine di policarbonato. Le conseguenze di questa decisione erano che non solo servivano motori lenti a velocità variabile, ma inoltre che il costo di produzione di un CD-ROM sarebbe potuto scendere al di sotto di un dollaro anche per quantità modeste.

Il Libro Giallo ha definito la formattazione dei dati, oltre a migliorare le capacità di correzione degli errori del sistema, un passo essenziale perché se agli amanti della musica non importa se si perde qualche bit qua e là, gli amanti del computer tendono ad essere Molto Sensibili al riguardo. Il formato base di un CD-ROM prevede di codificare ogni byte in un simbolo di 14 bit. Come abbiamo visto in precedenza, 14 bit sono sufficienti per codificare con il codice di Hamming un byte di 8 bit, con un avanzo di 2 bit; in pratica viene usato un sistema di codifica più potente. Il mapping da 14 a 8 bit in lettura viene fatto via hardware tramite una tabella di lookup.

Al livello successivo, un gruppo di 42 simboli consecutivi forma un frame di 588 bit; ogni frame contiene 192 bit di dati (ossia 24 byte). I 396 bit rimanenti sono usati come controllo e correzione degli errori. Fino a questo punto, lo schema dei CD audio e dei CD-ROM sembra essere identico.

Il Libro Giallo aggiunge il raggruppamento di 98 frame in un settore di CD-ROM, come mostrato in Figura 5.21; ogni settore di CD-ROM inizia con un preambolo di 16 byte, di cui i primi 12 sono 00FFFFFFFFFFFFF00 (esadecimale), per permettere al lettore di riconoscere l'inizio del settore del CD-ROM. I 3 byte successivi contengono il numero di settore, il quale è necessario perché l'operazione di posizionamento su una singola spirale di dati di un CD-ROM è molto più difficile di quanto non lo sia il posizionamento sulle tracce concentriche uniformi di un disco magnetico. Per il posizionamento, il software nel drive calcola approssimativamente dove andare, muove la testina fino al punto calcolato, e poi inizia a cercare nei dintorni un preambolo. Infine, l'ultimo byte del preambolo rappresenta il modo.

Il Libro Giallo definisce due modi: il modo 1 usa la rappresentazione di Figura 5.21, con un preambolo di 16 byte, 2048 byte di dati e 288 byte di codice di correzione di errore (un codice Reed-Solomon cross interleaved). Il modo 2 combina i dati e i campi di codici di correzione di errore in un campo dati di 2336 byte per quelle applicazioni che non hanno bisogno di correzione degli errori (o che non hanno il tempo di effettuarla), quali l'audio e il video. Si noti che per fornire alta affidabilità, vengono usati tre diversi schemi di correzione degli errori: all'interno di un simbolo, all'interno di un frame, e all'interno di un settore del CD-ROM. Errori del singolo bit vengono corretti al livello più basso, brevi

Fisicamente i CD-R cominciano con vuoti di policarbonato di 120 mm di diametro, come i CD-ROM, eccetto che contengono un ampio solco di 0,6 mm che serve a guidare il laser durante la fase di scrittura. Il solco ha un'escursione sinusoidale di 0,3 mm ad una frequenza di esattamente 22,05 kHz per fornire un feedback continuo, in modo che la velocità di rotazione sia controllata accuratamente, e in modificata se occorre. I CD-R sembrano CD-ROM, ma sono dorati invece che argentati. Il color oro proviene dal fatto che viene usato veramente dell'oro invece dell'alluminio sullo strato riflettente. A differenza dei CD argentati, i quali hanno delle depressioni fisiche su di essi, sui CD-R la diversa riflettività prodotta da fosse e terre deve essere simulata. Questo viene fatto aggiungendo uno strato di tinta tra il policarbonato e lo strato riflettente dorato, come mostrato in Figura 5.22. Esistono due tipi di tinta in uso: cianina, che è verde, e ftalocianina che è arancione giallastra. I chimici continuano a discutere su quale delle due sia meglio, queste tinte sono simili a quelle usate in fotografia, il che spiega perché Kodak e Fuji sono tra i principali produttori di CD-R vuoti.

Nel suo stato iniziale lo strato di tinta è trasparente, lascia passare la luce del laser che si riflette sullo strato dorato. Nella fase di scrittura il laser del CD-R viene alzato ad alta potenza (8-16mW). Quando il raggio colpisce la tinta, la scalda, rompendo un legame chimico; questo cambiamento nella struttura molecolare crea una macchia scura. In una successiva fase di lettura (a 0,5 mW), il fotorilevatore vede una differenza tra le macchie scure che sono state colpite dal laser, e le aree trasparenti che sono ancora intatte. Questa differenza è interpretata come una differenza tra fosse e terre, anche da un lettore CD-ROM normale o da un lettore audio CD.

Nessun tipo nuovo di CD potrebbe esistere senza avere associato alla propria esistenza un libro colorato e quindi il CD-R ha il **Libro Arancione**, pubblicato nel 1989. Questo documento definisce il CD-R, ed anche un nuovo formato il **CD-ROM XA** che permette ai CD-R di essere scritti in modo incrementale, alcuni settori oggi, alcuni domani e alcuni il mese prossimo. Un gruppo di settori consecutivi scritti in una volta viene detto **traccia del CD-ROM**.

Uno dei primi usi dei CD-R era per il Kodak PhotoCD. In questo sistema il cliente porta un rullino di pellicola fotosensibile e il suo PhotoCD nel negozio di sviluppo fotografie, e riceve il suo PhotoCD al quale sono state aggiunte le foto nuove, in coda a quelle già presenti sul CD. La nuova partita, che è stata creata scaunerizzando i negativi, viene scritta nel PhotoCD come una traccia separata sul CD-ROM. La scrittura incrementale era necessaria perché, quando il prodotto è stato introdotto, i CD-R vuoti erano troppo costosi per poterne usare uno nuovo per ogni rullino da sviluppare.

Comunque, la scrittura incrementale introduce un problema nuovo: prima del Libro Arancione, tutti i CD-ROM avevano un'unica VTOC (*tabella dei contenuti di volume*, Volume Table Of Contents) all'inizio. Questo schema non funziona nel caso di scrittura incrementale(multi-traccia).La soluzione presentata nel Libro Arancione è quella di dare ad ogni traccia del CD-ROM la sua VTOC; i file listati nella VTOC possono comprendere alcuni o tutti i file delle tracce precedenti. Dopo che il CD-R è stato inserito nel drive, il sistema operativo cerca in tutte le tracce del CD-ROM per localizzare la VTOC più recente, la quale contiene lo stato attuale del disco. Includendo alcuni, ma non tutti, i file delle tracce precedenti nella VTOC corrente, è possibile dare l'illusione che certi file siano stati cancellati. Le tracce sono raggruppate in **sessions**, e questo porta ai cosiddetti CD-ROM **multisessione**. Lettori audio CD standard non riescono a leggere i CD-ROM multisessione in quanto si aspettano un'unica VTOC all'inizio.

Ogni traccia deve essere scritta un'unica operazione continua, senza pause. Di conseguenza, il disco fisso da cui provengono i dati deve essere sufficientemente veloce da trasferirli in tempo. Se i file da copiare sono distribuiti in tutto il disco fisso, i tempi di posizionamento possono rallentare il flusso di dati che arriva al CD-R e produrre uno svu-

tamento del buffer. Questo ha l'effetto di produrre graziosi sottobicchieri (piuttosto costosi), o un frisbee giallo oro di 120 mm di diametro. Il software solitamente offre la possibilità di copiare tutti i file di input in un'unica immagine contigua del CD-ROM prima di iniziare la masterizzazione, ma questo processo effettivamente raddoppia il tempo di scrittura, richiede 650 Mb di spazio libero, e in ogni caso non protegge da eventuali situazioni di panico del disco fisso, che per esempio decide di iniziare la ricalibrazione termica perché si sta scalando troppo.

I CD-R danno la possibilità a privati e società di copiare facilmente i loro CD-ROM (e i CD audio), solitamente violando la legge sui copyright. Sono stati studiati diversi modi per rendere complicato questo tipo di pirateria, e per rendere difficile la lettura di un CD-ROM usando altri software che non siano quelli prodotti dal distributore. Uno di questi schemi consiste nel registrare tutte i file sul CD-ROM come se occupassero molti gigabyte, bloccando ogni tentativo di copiare i file sul disco fisso utilizzando metodi standard di copia di file. Le vere lunghezze dei file sono invece incluse nel software fornito dal distributore o nascoste (probabilmente criptate) nel CD-ROM in un qualche posto inatteso. Un altro schema, invece, usa codici di correzione dell'errore volutamente sbagliati all'interno di alcuni settori, basandosi sul fatto che il software che copia il CD corregga gli errori; il software applicativo controlla i codici di correzione errori, rifiutandosi di funzionare se non sono corretti. Altre possibilità consistono invece nell'utilizzare intervalli non standard tra tracce, ed altri difetti "fisici" del CD.

CD-riscrivibili

Anche se la gente è abituata a mezzi che possono essere scritti un'unica volta, quali la carta e le pellicole fotografiche, c'è richiesta di CD-ROM riscrivibili. Una tecnologia che oggi è disponibile è detta **CD-RW** (*CD-riscrivibile, CD-ReWritable*), che usa le stesse dimensioni di un CD-R. Invece di usare la tinta cianina o ftalocianina, i CD-RW usano una lega di argento, alluminio, indio, antimonio e tellurio per lo strato di registrazione. Questa lega ha due stati stabili: cristallino e amorfo, caratterizzati da diverse riflettività.

I drive dei CD-RW usano laser a tre diverse intensità: ad alta intensità il laser scioglie la lega, convertendola dallo stato cristallino ad alta riflettività nello stato amorfo a bassa riflettività che rappresenta le fosse; a media intensità la lega si scioglie e riforma il suo naturale stato cristallino e ridiventa terra; a bassa intensità viene "letto" lo stato del materiale, ma non avviene alcuna transizione di stato.

La ragione per cui i CD-RW non hanno sostituito i CD-R è legata al fatto che i CD-RW vuoti sono molto più costosi dei CD-R vuoti. Inoltre, per applicazioni che consistono nel fare il backup di dischi fissi, il fatto che una volta scritto, un CD-R non può essere cancellato per sbaglio, è un lato molto positivo.

DVD

Il formato base CD/CD-ROM è in circolazione dal 1980; da allora, la tecnologia è migliorata, al punto che dischi ottici di grande capacità sono oggi economicamente fattibili, e c'è una grande domanda per essi. Hollywood sarebbe molto felice di sostituire nastri analogici con dischi digitali, siccome i dischi hanno una qualità superiore, sono più economici da produrre, durano più a lungo, occupano meno spazio nei magazzini e non devono essere riavvolti. Le società di distribuzione di prodotti elettronici stanno cercando un nuovo prodotto di grande successo commerciale, e molte aziende di computer vogliono aggiungere caratteristiche multimediali ai loro software.

Questa combinazione di tecnologia e di richieste da parte di tre gruppi di industrie

immensamente ricche e potenti, ha portato al DVD, originariamente acronimo per *Disco Video Digitale* (Digital Video Disk), ma oggi ufficialmente per *Disco Versatile Digitale* (Digital Versatile Disk). I DVD sono basati sulle stesse idee di progettazione dei CD, sono formati da dischi di 120 mm di diametro in policarbonato, contenenti fosse e terre, illuminati da un diodo laser e letti da un fotoricettore. Di nuovo ci sono l'uso di:

1. Fosse più piccole (0,4 micron contro gli 0,8 dei CD).
2. Una spirale più stretta (0,74 micron tra una traccia e l'altra contro gli 1,6 micron dei CD).
3. Un laser rosso (a 0,65 micron contro gli 0,78 dei CD).

Questi miglioramenti aumentano la capacità del disco di sette volte, fino a 4,7 GB. Un DVD 1x opera a 1,4 Mb/s (contro i 150 Kb/s dei CD); sfortunatamente il passaggio al laser rosso significa che i lettori DVD dovranno essere dotati di una bizzarra ottica di conversione, oppure di un secondo laser, che permetta loro di leggere CD e CD-ROM, una cosa che non tutti i lettori DVD in commercio possiedono. Inoltre, leggere CD-R e CD-RW con un drive DVD può non essere possibile.

Sono abbastanza 4,7 GB? Forse. Usando la tecnica di compressione MPEG-2 (standardizzata col numero IS 13346) un DVD di 4,7 GB può contenere 133 minuti di video a pieno schermo, pieno moto e ad alta risoluzione (720x480), così come i dialoghi in al più otto lingue, e i sottotitoli in altre 32 lingue. Circa il 92 percento di tutti i film che Hollywood abbia mai prodotto sono al di sotto dei 133 minuti. Ciononostante, alcune applicazioni quali giochi multimediali o opere di riferimento (enciclopedie, eccetera) possono avere bisogno di più spazio, e Hollywood potrebbe voler mettere più di un film sullo stesso disco. A questo scopo sono stati definiti quattro formati:

1. Lato singolo, strato singolo (4,7 GB).
2. Lato singolo, strato doppio (8,5 GB).
3. Lato doppio, strato singolo (9,4 GB).
4. Lato doppio, strato doppio (17 GB).

Perché così tanti formati? In una parola: per politica. La Philips e la Sony volevano il lato singolo e strato doppio per le versioni ad alta capacità, mentre la Toshiba e la Time Warner volevano il lato doppio e strato singolo. Philips e Sony non pensavano che gli utenti

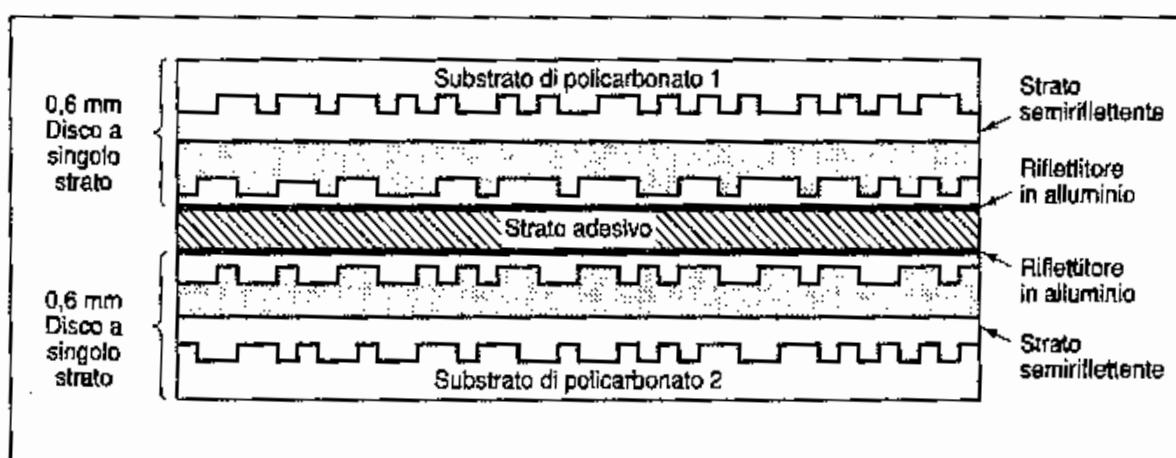


Figura 5.23 Un disco DVD doppio lato e doppio strato.

avessero voglia di girare il disco, mentre Time Warner non credeva che mettere due strati su di un lato avrebbe funzionato. Il compromesso raggiunto è stato quello di produrre tutte le combinazioni, e il mercato deciderà quale di queste sopravviverà.

La tecnologia a doppio strato ha uno strato riflettente sul fondo, ricoperto di uno strato semiriflettente; a seconda di dove il laser mette a fuoco, rimbalza su uno strato oppure sull'altro. Lo strato più basso ha bisogno di fosse leggermente più grandi per essere letto in modo affidabile, quindi la sua capacità è leggermente minore dello strato superiore.

I dischi a lato doppio sono costruiti usando due dischi a lato singolo di 0,6 mm e incollandoli schiena a schiena. Per rendere uguale lo spessore di tutti i dischi, i dischi singoli sono fatti da un disco di 0,6 mm incollato ad un substrato vuoto (o forse in futuro questo substrato sarà riempito con 133 minuti di pubblicità, nella speranza che la gente sia curiosa e vada a vedere cosa c'è dentro). La struttura del disco a doppio lato e doppio strato è descritta in Figura 5.23.

Il DVD è nato da un consorzio di 10 società di distribuzione di prodotti elettronici, sette delle quali giapponesi, in cooperazione con i maggiori studi cinematografici di Hollywood (alcuni dei quali sono in effetti di proprietà delle società giapponesi già presenti nel consorzio). Le industrie di computer e telecomunicazioni non erano state invitate alla festa, e lo scopo principale del DVD era quello di usarlo per il noleggio di film; ad esempio, tra le caratteristiche tipiche si trovano il salto in real-time di scene spinte (in modo da permettere ai genitori di trasformare un film vietato ai minori in uno adatto per i loro bambini), suono a sei canali e supporto per il Pan-and-Scan. Quest'ultima proprietà permette al lettore DVD di decidere in modo dinamico in che modo tagliare i bordi destro e sinistro dei film (il cui rapporto larghezza:altezza è 3:2) in modo da adattarsi ai moderni apparecchi televisivi (il cui rapporto è 4:3).

Un'altra caratteristica a cui l'industria del computer non avrebbe pensato è quella di una deliberata incompatibilità tra dischi prodotti per il mercato statunitense e dischi prodotti per il mercato europeo, ed ancora altri standard per altri continenti. Questa caratteristica è stata richiesta da Hollywood, in quanto i film nuovi sono sempre distribuiti prima negli Stati Uniti, poi spediti in Europa quando i video stanno già uscendo negli Stati Uniti. L'idea è quella di fare in modo che i negozi di video europei non comprino i video dagli Stati Uniti troppo presto, riducendo il successo in sala dei film sul mercato europeo. Se Hollywood gestisse l'industria del computer, avremmo floppy da 3,5 pollici negli Stati Uniti e da 9 centimetri in Europa.

5.4.2 La formattazione di un disco

Un disco fisso si compone di una pila di piatti in alluminio, lega, o di vetro, di diametro pari a 5,25 o 3,5 pollici (circa 13 o 9 cm rispettivamente; anche meno su un computer portatile). Su ogni piatto è depositato un sottile strato di ossido di metallo magnetizzabile. Sul disco non è posta alcuna informazione in fabbrica.

Prima che il disco possa essere usato, ogni piatto deve ricevere una formattazione di basso livello, fatta via software. La formattazione consiste in una serie di tracce concen-

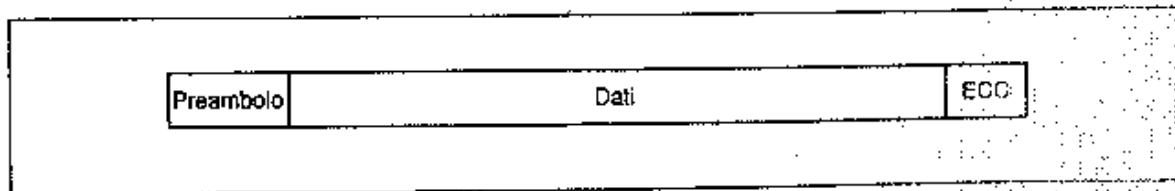


Figura 5.24 Un settore di disco.

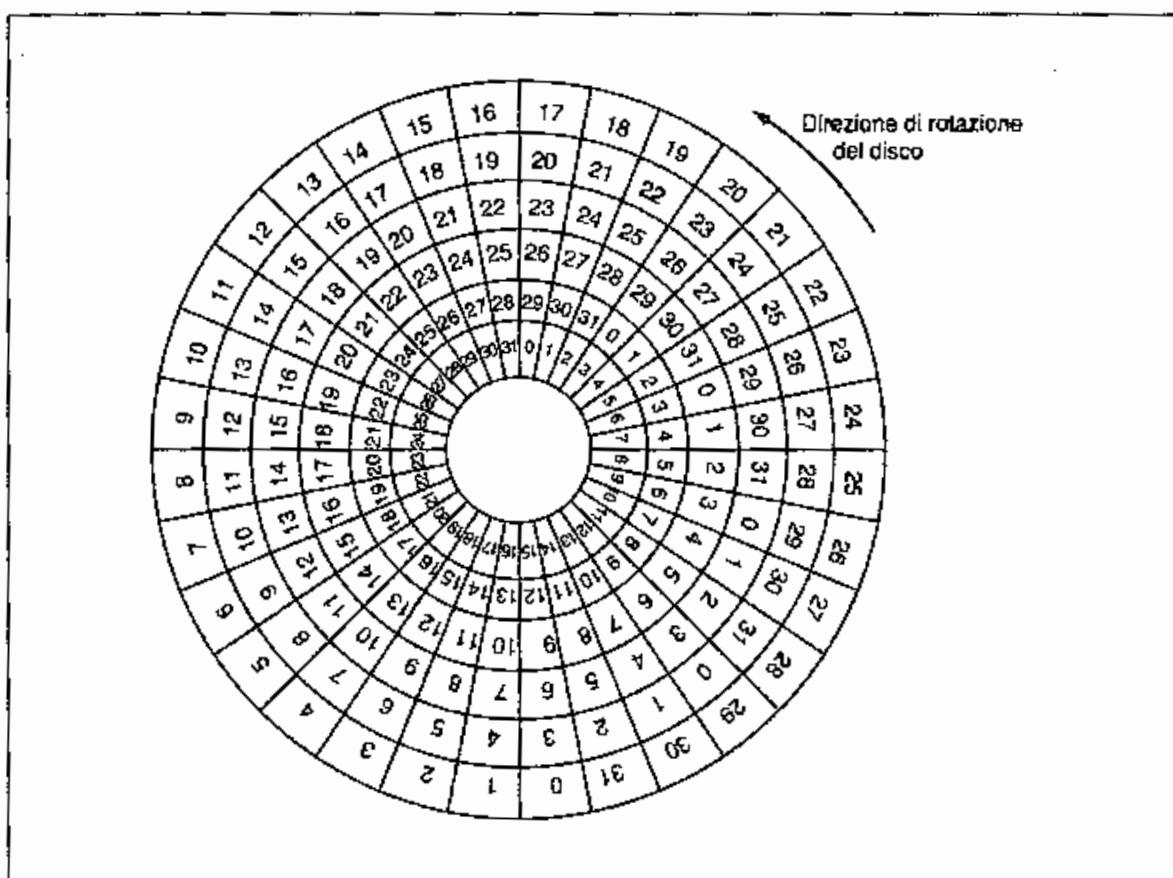


Figura 5.25 Illustrazione della pendenza del cilindro.

triche, ognuna contenente alcuni numeri di settore, con piccoli intervalli tra i settori. Il formato di un settore è mostrato in Figura 5.24.

Il preambolo comincia con un certo pattern di bit che permette all'hardware di riconoscere l'inizio di un settore; contiene inoltre i numeri di cilindri e settori e qualche altra informazione. La dimensione della porzione di dati è determinata dal programma di formattazione di basso livello, ma la maggior parte dei dischi usa settori da 512 byte. Il campo ECC contiene informazioni ridondanti che possono essere utilizzate per ripristinare il disco dopo un errore in lettura. La grandezza e il contenuto di questo campo variano da produttore a produttore, a seconda di quanto spazio di disco il progettista è disposto a "sprecare" per ottenere una più alta affidabilità, e della complessità del codice ECC che il controllore può gestire: un ECC a 16 byte non è insolito. Inoltre, tutti i dischi fissi hanno un certo numero di settori in più, pronti per essere usati al posto di eventuali settori con difetti di fabbricazione.

Facendo la formattazione a basso livello, la posizione del settore 0 in ogni traccia si sposta di una certa distanza rispetto al settore 0 della traccia precedente. Questa distanza è detta **pendenza del cilindro** (cylinder skew), e viene fatta per aumentare le prestazioni: l'idea è quella di fare il modo che il disco legga più tracce in un'operazione continua, senza perdere dati. La natura del problema può essere compresa, analizzando la Figura 5.18(a); si supponga che una richiesta abbia bisogno di accedere a 18 settori, a partire dal settore 0 nella traccia più interna. Lleggere i primi 16 settori richiede una rotazione del disco, ma per leggere il settore 17 è necessaria un'operazione di posizionamento sulla traccia più esterna. Quando la testina è riuscita a muoversi di una traccia, il settore

0 ha superato la testina, così è necessario un giro completo perché il settore 0 ripassi sotto la testina. Il problema è eliminato spostando lo stesso settore in tracce diverse, come mostrato in Figura 5.25.

L'ammontare di questa pendenza del cilindro dipende dalla geometria del drive; per esempio, un drive da 10000 giri al minuto (rpm, rounds per minute) ruota in 6 ms. Se una traccia contiene 300 settori, un nuovo settore passa sotto la testina ogni 20 µs, quindi, se il tempo di posizionamento da traccia a traccia è di 800 µs, passeranno 40 settori sotto la testina durante il posizionamento, quindi la pendenza del disco dovrebbe essere di almeno 40 settori, invece dei 3 settori mostrati in Figura 5.25. È il caso di dire che anche il passaggio da una testina all'altra richiede un tempo finito, quindi esiste anche una pendenza della testina, oltre a quella del cilindro, ma questa non è molto grande.

Come risultato della formattazione a basso livello, la capacità del disco si riduce, in relazione alle dimensioni del preambolo, dello spazio tra settori e ECC, così come del numero di settori lasciati di riserva: spesso la capacità di un disco formattato è del 20% minore di quella del disco originale. I settori di riserva non contano nel computo della capacità, quindi tutti i dischi di un certo tipo che lasciano la fabbrica hanno esattamente la stessa capacità, indipendentemente dal numero di settori danneggiati che contengono (se il numero di settori danneggiati supera il numero di settori di ricambio, il disco sarà eliminato e non verrà distribuito).

C'è una certa confusione sulla capacità dei dischi, perché alcuni produttori pubblicizzano la capacità del disco non formattato, in modo che i loro drive sembrino più grossi di quanto realmente non siano. Per esempio consideriamo un drive la cui capacità da non formattato sia 20×10^9 byte. Questo può essere venduto come un disco da 20 GB, comunque dopo la formattazione forse solo $2^{34} = 17,2 \times 10^9$ byte sono ancora disponibili per i dati. Per aggiungere ulteriore confusione, il sistema operativo probabilmente riporterà una capacità di 16 GB, e non di 17,2, in quanto il software considera 1 GB come 2^{30} (1 073 341 824) byte e non 10^9 nessia 1 000 000 000 byte.

Per rendere le cose ancora più complicate, nel mondo della comunicazione, 1 Gbps significa 1 000 000 000 bit al secondo, dato che il prefisso *giga* significa proprio 10^9 (un chilometro è 1000 metri e non 1024, dopo tutto). Solo nel caso di memoria e dimensioni dei dischi chilo, mega, giga e tera significano 2^{10} , 2^{20} , 2^{30} e 2^{40} , rispettivamente.

La formattazione influenza anche le prestazioni: se un disco da 10 000 giri al minuto ha 300 settori per traccia da 512 byte a testa, servono 6 ms per leggere i 153 600 byte di una traccia, ad una frequenza di 25 600 000 byte al secondo, o 24,4 MB al secondo. Non è possibile andare più velocemente, indipendentemente dall'interfaccia, anche se questa è un'interfaccia SCSI a 80MB/s o 160MB/s.

In effetti, letture continue a queste frequenze richiedono controllori con ampi buffer. Si consideri, per esempio, un controllore con un buffer di un settore, che abbia ricevuto il comando di leggere due settori consecutivi: dopo la lettura del primo settore dal disco e il calcolo dell'ECC, i dati devono essere trasferiti in memoria centrale e, durante questo trasferimento, il settore successivo passerà sotto la testina. Quando la copia in memoria è completa, il controllore dovrà aspettare quasi un intero tempo di rotazione prima che il secondo settore passi nuovamente.

Questo problema può essere eliminato numerando i settori in modo interallacciato, durante la formattazione del disco. In Figura 5.26(a) vediamo la solita numerazione (in questo caso abbiamo ignorato la pendenza del cilindro), mentre in Figura 5.26(b) vediamo il singolo interleaving (interallacciamento), che lascia al controllore un po' di respiro tra un settore e l'altro, in modo da copiare il buffer in memoria centrale.

Se il processo di copiatura è molto lento, può risultare utile il doppio interleaving di Figura 5.26(c). Se il controllore ha un buffer di un solo settore, non ha importanza che la copia dal buffer in memoria venga fatta dal controllore, dalla CPU o da un

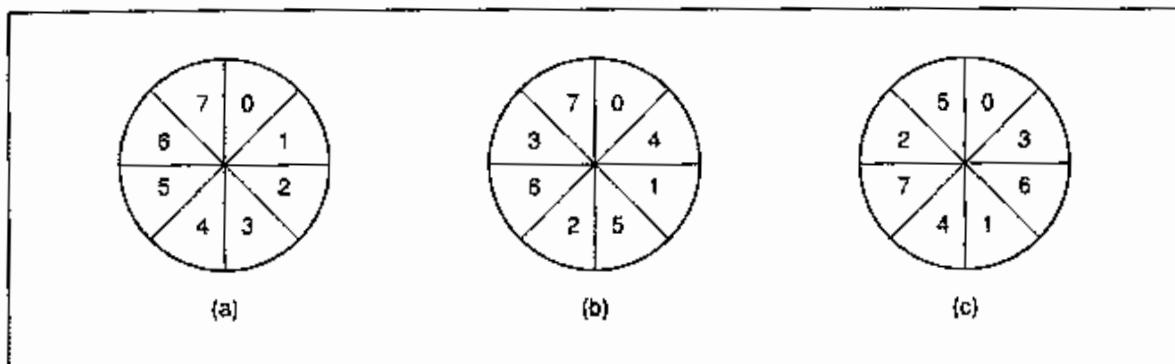


Figura 5.26 (a) Nessun interallacciamento. (b) Singolo interallacciamento. (c) Doppio interallacciamento.

DMA: richiederà comunque un certo tempo. Per evitare l'interleaving, il controllore dovrebbe riuscire a bufferizzare un'intera traccia; molti controllori moderni sono in grado di farlo.

Dopo il completamento della formattazione di basso livello, il disco viene partizionato: dal punto di vista logico, ogni partizione è un disco diverso. Sul Pentium e sulla maggior parte dei calcolatori moderni, il settore 0 contiene il **master boot record** (record di avvio principale), che contiene del codice di avvio e la tabella delle partizioni. Questa tabella contiene il settore di inizio e la dimensione di ogni partizione. Sul Pentium, la tabella delle partizioni ha spazio per quattro partizioni; se sono tutte per Windows, si chiameranno C:, D:, E: e F: e trattate come drive separati. Se invece tre di esse sono occupate da Windows e una da UNIX, allora Windows chiamerà queste partizioni C:, D: ed E:, mentre il primo CD-ROM sarà F:. Per riuscire ad avviare il sistema dal disco fisso, una partizione deve essere marcata attiva nella tabella delle partizioni.

Il passo finale nella preparazione del disco è una **formattazione ad alto livello** di ogni partizione (separatamente). Questa operazione aggiunge un blocco di avvio, la gestione dello spazio libero (lista o bitmap), la directory radice e un file system vuoto. Inoltre mette un codice nella tabella delle partizioni relativa al disco in questione, che indica quale file system viene usato nella partizione, in quanto molti sistemi operativi supportano file system multipli e fra loro incompatibili (per motivi storici). A questo punto il sistema può essere avviato.

Quando si accende il calcolatore, il BIOS inizia l'esecuzione e poi va a leggere il master boot record ed effettua un'operazione di jump su di esso. Il programma di avvio controlla quale partizione è attiva, legge il settore di avvio di tale partizione, e lo esegue. Il settore di avvio contiene un piccolo programma che cerca nella directory radice un particolare programma (o il sistema operativo o un programma di avvio più grande), il quale viene caricato in memoria ed eseguito.

5.4.3 Gli algoritmi di schedulazione del braccio del disco

In questo paragrafo si esamineranno alcune delle caratteristiche generali del driver del disco. Per prima cosa, consideriamo il tempo necessario a leggere o a scrivere un blocco sul disco; esso è determinato da tre fattori:

1. Il tempo di posizionamento della testina (*seek time*: il tempo necessario a spostare la testina di lettura sul cilindro richiesto).

2. Il ritardo di rotazione (il tempo necessario al settore richiesto per passare sotto alla testina di lettura).
3. Il tempo di trasferimento dei dati vero e proprio.

Per la maggior parte dei dischi, il tempo di posizionamento è quello predominante, per cui ridurre il tempo medio di seek può significare aumentare sostanzialmente le prestazioni del sistema.

Se il driver del disco accetta i comandi uno alla volta, e li esegue in quell'ordine, cioè First-Come First-Served (FCFS, primo arrivato, primo servito), non si può fare molto per ottimizzare il tempo di posizionamento; ma quando il disco lavora parecchio, è possibile adottare un'altra strategia. È probabile che, mentre il braccio delle testine si sta spostando in conseguenza di una richiesta, altri processi generino richieste di operazioni sul disco. Molti driver di dischi mantengono una tabella, indirizzata dal numero di cilindro, i cui elementi puntano a liste concatenate contenenti le richieste pendenti relative ad ogni cilindro.

Con una simile struttura dati si può migliorare l'algoritmo di schedulazione FCFS; per vedere questo miglioramento, si consideri un disco con 40 cilindri. Arriva una richiesta di lettura di un blocco che sta sul cilindro 11; mentre la testina sta posizionandosi sul cilindro 11, arrivano nuove richieste per i cilindri 1, 36, 16, 34, 9 e 12, in quest'ordine. Esse sono memorizzate nella tabella delle richieste pendenti, in una lista separata per ogni cilindro. Le richieste sono mostrate in Figura 5.27.

Quando la richiesta corrente (per il cilindro 11) è stata completata, il driver del disco deve scegliere quale sarà la prossima richiesta da gestire. Usando la politica FCFS, dovrebbe prima far spostare la testina sul cilindro 1, poi sul 36 e così via; questo algoritmo richiederà lo spostamento del braccio per 10, 35, 20, 18, 25 e 3 cilindri, rispettivamente, per un totale di 111 cilindri.

In alternativa, potrebbe sempre scegliere di eseguire l'operazione relativa alla richiesta più vicina, per minimizzare il tempo di posizionamento. Date le richieste della Figura 5.27, la sequenza è 12, 9, 16, 1, 34 e 36, come mostrato dalla linea spezzata sotto la Figura 5.27. Per questa sequenza, gli spostamenti del braccio sono di 1, 3, 7, 15, 33 e 2 cilindri, per un totale di 61 cilindri. Questo algoritmo, detto Shortest Seek First (SSF, prima il cammino più breve), riduce di circa la metà gli spostamenti del braccio, rispetto all'algoritmo FCFS.

Sfortunatamente, l'algoritmo SSF ha un problema. Si supponga che arrivino altre richieste mentre vengono elaborate quelle della Figura 5.27; per esempio, se dopo essere

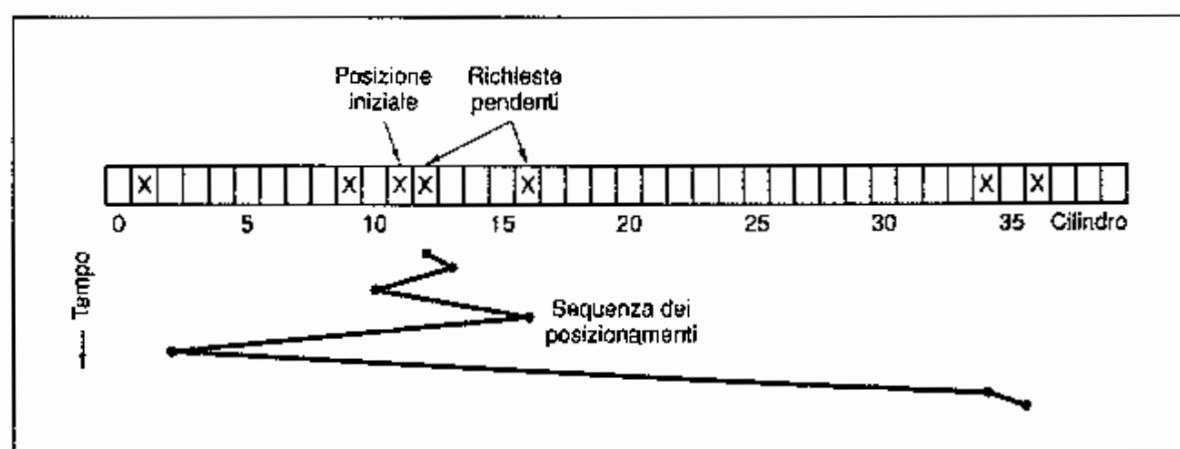


Figura 5.27 Algoritmo di schedulazione del braccio detto Shortest Seek First (SSF, cammino di posizionamento più breve per primo).

andati al cilindro 16 si presenta una nuova richiesta per il cilindro 8, questa avrà priorità sulla richiesta relativa al cilindro 1. Se a questo punto si presenta una richiesta per il cilindro 13, il braccio si sposterà sul cilindro 13, anziché sul cilindro 1. Nel caso il disco sia usato molto pesantemente, il braccio tenderà a rimanere sempre nella parte centrale del disco, così che le richieste relative ai cilindri molto esterni o molto interni si troveranno a dover aspettare, per essere servite, che una oscillazione statistica del carico faccia sì che non vi siano richieste per i cilindri interni. Possiamo dire, quindi, che le richieste lontane dalla metà del disco ricevono un servizio poco efficiente; in questo caso il tempo di risposta minimo e l'equità dell'algoritmo sono in conflitto fra di loro.

Anche i palazzi alti hanno a che fare con un conflitto analogo: il problema di schedulazione di un ascensore in un grattacielo è simile al problema della schedulazione del braccio delle testine. Le richieste arrivano di continuo, chiamando l'ascensore in maniera casuale ad uno dei piani (cilindri) dell'edificio, e il microprocessore che controlla l'ascensore potrebbe facilmente tenere traccia dell'ordine in cui gli utenti chiamano l'ascensore dai vari piani e servirsi di una politica FCFS o SSF.

La maggior parte degli ascensori, usa invece un algoritmo diverso per riconciliare l'equità del servizio con l'efficienza: essi iniziano a muoversi in una direzione, e la cambiano solo quando in quella direzione non vi sono più richieste pendenti. Questo algoritmo, noto sia nel mondo dei dischi, che in quello degli ascensori, con il nome di **algoritmo dell'ascensore**, richiede al software di mantenere un unico bit di informazione: il bit della direzione corrente, UP (su) o DOWN (giù). Quando una richiesta è stata soddisfatta, il disco o l'ascensore controllano questo bit. Se è messo a UP, il braccio o l'ascensore si muovono verso la prossima richiesta più alta in attesa, sempre che ce ne siano. Se nessuna richiesta più alta è in attesa, il bit di direzione viene invertito. Quando il bit di direzione è messo a DOWN, il movimento avviene verso la prossima richiesta pendente più bassa, sempre che ce ne siano.

La Figura 5.28 mostra l'algoritmo dell'ascensore con le stesse richieste della Fig. 5.27, supponendo che la direzione iniziale sia UP. L'ordine in cui sono serviti i cilindri è 12, 16, 34, 36, 9 e 1, il che comporta uno spostamento del braccio delle testine di 1, 4, 18, 2, 27 e 8 cilindri, per un totale di 60 cilindri. In questo caso l'algoritmo dell'ascensore è leggermente migliore di quello SSF, anche se, nel caso medio, è peggiore. Una proprietà piacevole di questo algoritmo è che, dato un qualsiasi insieme di richieste, la quantità massima di cilindri di cui deve essere spostato il braccio per servire le richieste, è fissa: il doppio del numero dei cilindri.

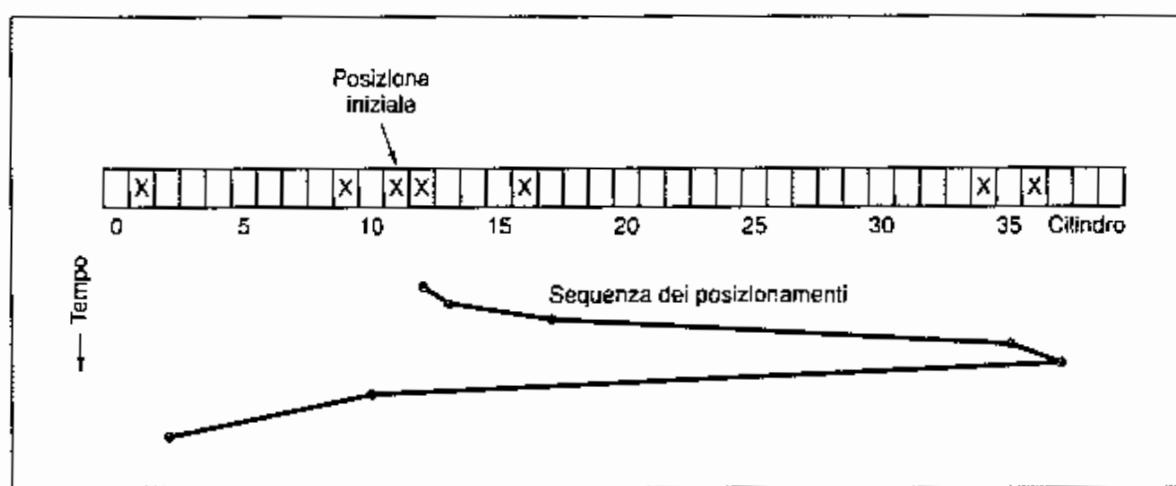


Figura 5.28 L'algoritmo dell'ascensore per la schedulazione delle richieste al disco.

Una piccola modifica di questo algoritmo, che comporta una minore varianza nei tempi di risposta (Teory, 1972), è quella di esaminare il disco sempre nella stessa direzione. Quando è stata servita la richiesta pendente relativa al cilindro con numero più alto, il braccio si sposta sul cilindro col numero più piccolo per il quale sia presente una richiesta, e continua a muoversi nella direzione UP. In effetti, è come se il cilindro con il numero più piccolo fosse considerato l'immediato successore di quello con il numero più grande.

Alcuni controllori di dischi danno la possibilità al software di conoscere il numero del settore sul quale si trova la testina; quando si ha a disposizione uno di questi controllori, è possibile procedere ad un'ulteriore ottimizzazione. Se vi sono due o più richieste in attesa per lo stesso cilindro, il driver può inviare una richiesta per il settore che passerà per primo sotto la testina di lettura/scrittura. Si noti che, nel caso si abbiano più tracce per cilindro, si possono avere richieste consecutive per tracce diverse senza alcun degrado delle prestazioni: questo perché il controllore può selezionare una qualunque delle sue testine in maniera istantanea, dato che la selezione di una testina non implica né spostamenti del braccio, né ritardi di rotazione.

Se il disco ha la proprietà che il tempo di posizionamento è molto minore del ritardo di rotazione, allora si deve usare una strategia di ottimizzazione diversa. Le richieste pendenti devono essere ordinate per numero di settore, ed appena il settore successivo sta per passare sotto la testina, il braccio deve essere messo sulla traccia giusta, per effettuare la lettura o la scrittura.

Con i dischi fissi moderni, i ritardi di posizionamento e rotazione dominano le prestazioni al punto che leggere uno o due settori alla volta è molto inefficiente; per questa ragione, molti controllori di disco leggono e mettono in cache più settori, anche se c'è un'unica richiesta. Tipicamente, ogni richiesta di lettura di un settore produrrà la lettura di quel settore e della maggior parte o di tutta la traccia che lo contiene, a seconda di quanto sia lo spazio disponibile nella memoria cache del controllore. Il disco descritto in Figura 5.17 ha 2 Mb o 4 Mb di cache, per esempio. L'uso della cache è determinato dinamicamente dal controllore; nella sua modalità più semplice, la cache è divisa in due sezioni, una per le letture e una per le scritture; se una lettura successiva può venire soddisfatta tramite la cache del controllore, i dati richiesti possono essere restituiti immediatamente.

È il caso di notare che la cache del controllore di disco è completamente indipendente dalla cache del sistema operativo. La cache del controllore solitamente contiene dei blocchi che in effetti non sono stati richiesti, ma che era facile leggere, in quanto sono passati sotto la testina come effetto collaterale di una qualche altra lettura. Per contro, una qualunque cache mantenuta dal sistema operativo si compone di blocchi che sono stati letti in modo esplicito, e che il sistema operativo pensa possano essere utili in un futuro prossimo (ad esempio, un blocco di disco contenente un blocco di directory).

Quando è presente più di un dispositivo fisico sullo stesso controllore, è necessario mantenere una tabella delle richieste pendenti separata per ognuno dei dischi. Ogni volta che un disco è inutilizzato, si può mandare un comando di spostamento del braccio verso il cilindro necessario per completare la prossima richiesta pendente (supponendo che il controllore permetta operazioni di spostamento multiple). Quando finirà il trasferimento in corso, si può effettuare un controllo per vedere se uno qualunque dei dischi si trovi già posizionato sul cilindro giusto. Se uno o più dischi sono già posizionati correttamente, allora si può far partire il trasferimento relativo ad uno di questi dischi; se nessun braccio si trova sul cilindro giusto, il driver dovrà mandare un nuovo comando di spostamento sul disco che ha appena terminato il trasferimento, ed aspettare fino alla prossima interruzione per vedere quale braccio sia arrivato per primo a destinazione.

È importante rendersi conto che tutti gli algoritmi di schedulazione del disco appena descritti, suppongono tacitamente che la geometria reale del disco corrisponda alla geometria virtuale. Se questo non è vero, allora le richieste di schedulazione del disco non

hanno senso, in quanto il sistema operativo non può sapere quale, tra il cilindro 40 e il cilindro 200, sia veramente il più vicino al cilindro 39. D'altra parte, se il controllore del disco può accettare più richieste in sospeso, allora può usare questi algoritmi di schedulazione internamente; in questo caso, gli algoritmi sono ancora validi, ma ad un livello più interno, ossia nel controllore.

5.4.4 Trattamento degli errori

I produttori di dischi spingono costantemente i limiti della tecnologia aumentando la densità lineare dei bit. Una traccia a metà strada di una disco a 5,25 pollici ha una circonferenza di circa 300 mm; se contiene 300 settori di 512 byte, la densità di registrazione lineare può essere di circa 5000 bit/mm, tenendo conto dello spazio perso con i preamboli, gli ECC e gli spazi tra settori. Registrare 5000 bit/mm richiede un substrato estremamente uniforme e una copertura di ossido molto buona, ma sfortunatamente non è possibile costruire un disco che segua tali specifiche e sia privo di difetti. Non appena la tecnologia di manifattura dei dischi migliorerà al punto in cui sarà possibile operare senza problemi a tali densità, i progettisti di dischi aumenteranno la densità per migliorare la capacità del disco; e probabilmente in questo modo reintrodurranno i difetti.

I difetti di produzione introducono settori corrotti, ossia settori che non sono in grado di restituire in modo corretto un valore che è stato scritto su di essi. Se il difetto è molto piccolo, ad esempio di pochi bit, è comunque possibile usare il settore corrotto e lasciare che l'ECC corregga gli errori ogni volta; se il difetto è maggiore, l'errore non può essere mascherato.

Esistono due approcci generali ai settori corrotti: gestirli a livello del controllore, o gestirli a livello di sistema operativo. Nel primo modo, prima che il disco esca dalla fabbrica, esso viene collaudato, e i settori corrotti vengono scritti in una lista sul disco stesso; ogni settore corrotto viene sostituito da uno di quelli di riserva.

Questa sostituzione può essere effettuata in due modi diversi. In Figura 5.29(a) vediamo una singola traccia contenente 30 settori dati e 2 settori di riserva; il settore 7 è difettoso. Il controllore può rimappare uno dei settori di riserva nel settore 7, come in Figura 5.29(b), oppure può spostare tutti i settori avanti di uno, come mostrato in Figura 5.29(c). In entrambi i casi, il controllore deve saper distinguere i settori: può tenere traccia di queste informazioni in tabelle interne (una per traccia) o riscrivendo i preamboli in modo da rispecchiare i numeri di settore rimappati. Se vengono modificati i preamboli, il metodo

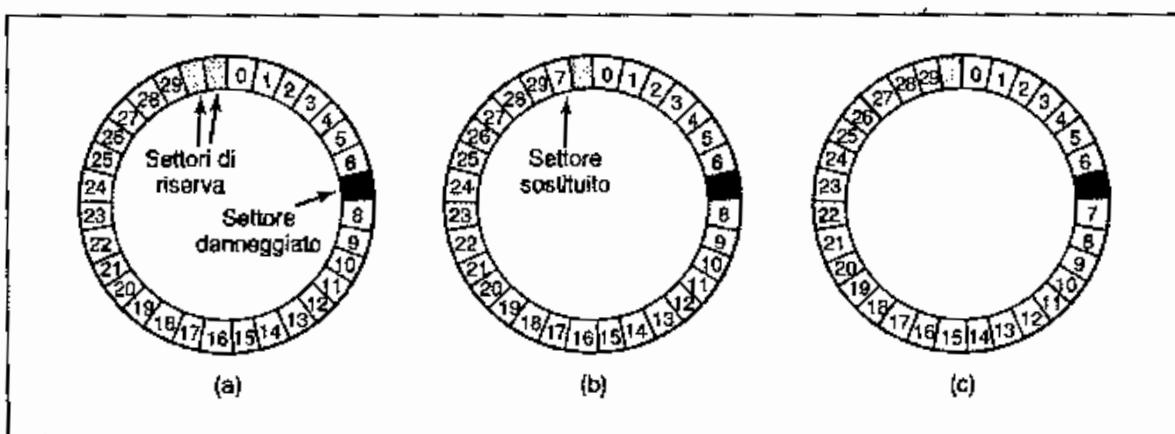


Figura 5.29 (a) Una traccia con un settore rovinato. (b) Sostituzione di un settore di riserva con quello rovinato. (c) Spostamento di tutti i settori per ignorare quello rovinato.

di Figura 5.29(c) è più dispendioso, in quanto richiede la riscrittura di 23 preamboli, ma alla fine fornisce prestazioni migliori in quanto l'intera traccia può essere ancora letta in un'unica rotazione.

Vi sono errori che possono anche prodursi durante operazioni normali, dopo che il drive è stato installato. La prima linea di difesa contro errori che l'ECC può non riuscire a gestire è quella di ripetere la lettura: alcuni errori sono transitori, ossia sono causati da granelli di polvere sotto la testina, e spariranno ad un secondo tentativo. Se il controllore si accorge che si ripetono gli errori su un particolare settore, può attivare un settore di riserva prima che il settore in questione diventi totalmente inutilizzabile. In questo modo non c'è perdita di dati e sia il sistema operativo sia l'utente non si accorgono di nulla. Solitamente si deve usare il modo descritto in Figura 5.29(b) in quanto gli altri settori della traccia possono già contenere dati, ed usare il metodo di Figura 5.29(c) richiederebbe non solo la riscrittura dei preamboli, ma la copiatura di tutti i dati.

Abbiamo detto che esistono due approcci generali alla gestione degli errori: gestirli dal lato del controllore o dal lato del sistema operativo; se il controllore non ha la capacità di rimappare i settori in modo trasparente, come abbiamo appena descritto, il sistema operativo deve svolgere questo compito, via software. Questo significa che deve prima di tutto acquisire una lista dei settori corrotti, o leggendola da disco o semplicemente controllando il disco. A questo punto può costruire delle tabelle di rimappatura; se il sistema operativo vuole usare il metodo descritto in Figura 5.29(c), deve spostare i dati dal settore 7 al 29 di una posizione in avanti.

Se il sistema operativo gestisce la rimappatura, deve assicurarsi che non ci siano settori corrotti contenenti parti di file, né contenenti la lista o la bitmap dei blocchi liberi; un modo per fare ciò è tenere un file segreto contenente la lista dei settori rovinati: questo file non viene inserito nel file system, così gli utenti non lo possono leggere accidentalmente (o, ancora peggio, non possono cancellarlo).

In ogni modo esiste ancora un altro problema, quello dei backup. Se si effettua il backup file per file, è importante che la funzione di utilità non cerchi di copiare il file dei settori corrotti; per impedirlo il sistema operativo deve nascondere tale file in modo che nemmeno l'utilità di backup possa trovarlo. Se il backup viene effettuato settore per settore invece che file per file, sarà difficile, se non impossibile, impedire errori di lettura durante il backup. L'unica speranza è che il programma di backup sia abbastanza furbo da arrendersi dopo 10 tentativi di lettura falliti, e da proseguire col settore successivo.

I settori corrotti non sono l'unica fonte di errore; si possono ad esempio verificare errori di posizionamento dovuti a problemi meccanici del braccio. Il controllore tiene traccia internamente delle posizioni del braccio; per effettuare un posizionamento, emette una serie di impulsi verso il motore del braccio, un impulso per cilindro, per forzare il posizionamento su un nuovo cilindro. Quando il braccio meccanico arriva alla sua destinazione finale, il controllore legge il numero effettivo di cilindro dal preambolo del primo settore incontrato: se il braccio è nella posizione sbagliata, si è verificato un errore di posizionamento.

La maggior parte dei controllori di disco fisso corregge gli errori di posizionamento in modo automatico, ma la maggior parte dei controllori di dischetti (tra cui quelli dei Pentium) non fa altro che impostare un bit di errore e lascia il controllo al driver. Il driver tratta questa condizione mandando un comando di recalibrate, per spostare il braccio fino a che può essere spostato, e obbligare il controllore a considerare il cilindro corrente come il cilindro 0. Di solito, queste operazioni sono sufficienti per risolvere il problema, altrimenti, il disco deve essere riparato.

Come si è visto, il controllore non è altro che un piccolo calcolatore specializzato, completo di software, variabili, buffer e, qualche volta, anche di errori. Qualche volta, una sequenza insolita di eventi, come per esempio un'interruzione da un disco contemporaneamente a un'altra, può causare un errore.

nea ad un comando `recalibrate` su di un altro disco, può rendere evidente un baco, e può far sì che il controllore entri in loop, o comunque perda il controllo di ciò che stava facendo. I progettisti dei controllori, di solito, prevedono il peggio, e mettono a disposizione un piedino del circuito integrato che, se impostato, fa sì che il controllore dimentichi qualsiasi cosa stia facendo e si reinizializzi. Se tutte le altre operazioni fallissero, il driver del disco potrebbe impostare un bit per attivare questo segnale e reinizializzare il controllore. Se neanche questa operazione avesse successo, il driver non potrebbe far altro che stampare un messaggio e terminare.

La ricalibrazione di un disco produce uno strano rumore, ma a parte questo è un'operazione che non disturba; ci sono però dei casi in cui la ricalibrazione diventa un problema serio: in sistemi con vincoli di tempo reale. Quando si mostra un video salvato su disco fisso, o si vogliono copiare dei file su CD-ROM dal disco fisso, è essenziale che i bit arrivino dal disco fisso a frequenza uniforme. In questi casi, la ricalibrazione inserisce degli spazi nel flusso di bit e quindi non è accettabile: drive speciali, chiamati **dischi AV** (AudioVisivi), che non si ricalibrano mai, sono disponibili per questo tipo di applicazioni.

5.4.5 Memorizzazione stabile

Come abbiamo visto, i dischi a volte commettono degli errori. Settori buoni possono improvvisamente diventare settori rovinati, interi drive possono morire in modo inaspettato. I sistemi RAID proteggono contro la situazione in cui alcuni settori diventino inutilizzabili, o anche contro la situazione in cui un drive non sia più leggibile, comunque non proteggono da errori in scrittura che salvino dati sbagliati; inoltre, non proteggono da crash durante una scrittura che rovinano i dati originali, e non effettuano la sostituzione con i dati nuovi.

Per alcune applicazioni, è essenziale che i dati non siano mai persi né corrotti, anche nel caso di errori di disco o di CPU. Idealmente, un disco dovrebbe funzionare continuamente, senza errori, ma sfortunatamente questo non è possibile. Quello che si può ottenere è un sottosistema del disco con la seguente proprietà: quando si effettua una scrittura sul disco, o questa scrittura avviene in modo corretto, o non avviene affatto, in modo da lasciare intatti i dati esistenti. Questo sistema è chiamato **memorizzazione stabile**, ed è implementato in software (Lampson e Sturgis, 1979). Nel seguito descriveremo una leggera variazione dell'idea originale.

Prima di descrivere l'algoritmo, è importante avere un modello chiaro degli errori possibili. Il modello suppone che quando un disco scrive in un blocco (uno o più settori), o la scrittura avviene correttamente, o non è corretta, e che se questo accade, l'errore è identificabile in una lettura successiva esaminando i valori dei campi ECC. In linea di principio, un rilevamento degli errori garantito è impossibile perché con, ad esempio, un campo ECC di 16 byte per un settore di 512 byte, ci sono 2^{1096} valori per i dati e solo 2^{144} valori di ECC. Pertanto se un blocco è danneggiato in una scrittura ma il campo ECC è corretto, ci sono miliardi e miliardi di combinazioni non corrette che possono portare allo stesso ECC; se una di esse si verifica, l'errore non verrà rilevato. Nell'insieme, la probabilità che dati casuali producano i 16 bit dell'ECC giusti è di circa 2^{-144} , che è sufficientemente piccolo da poter essere chiamato zero, anche se non lo è.

Il modello ipotizza anche che un settore scritto correttamente possa rovinarsi spontaneamente, e diventare impossibile da leggere. Comunque, si suppone che questi eventi siano così rari, che la probabilità che lo stesso settore si rovini anche su un secondo drive (indipendente) durante un intervallo di tempo ragionevole (ad esempio, 1 giorno) è abbastanza piccola da venire ignorata.

Il modello suppone anche che la CPU possa fallire, nel qual caso si ferma; si ferma

anche ogni scrittura del disco in progresso al momento del fallimento, il che produce dati non corretti all'interno di un settore e ad un ECC non corretto che può quindi essere rilevato. In tutte queste ipotesi, la memorizzazione stabile può essere resa affidabile al 100% nel senso che le operazioni di scrittura, o funzionano correttamente, o lasciano invariati i vecchi dati. Naturalmente, non protegge contro disastri fisici, tipo terremoti che fanno cadere il computer da 100 metri di altezza dentro un pentolone di magma bollente. In effetti è abbastanza difficile riuscire a recuperare da siffatte situazioni via software.

La memorizzazione stabile usa una coppia di dischi identici dove i blocchi corrispondenti lavorano insieme per formare un blocco libero da errori; in assenza di errori i blocchi corrispondenti di entrambi i dischi sono uguali e possono essere letti indifferentemente ottenendo lo stesso risultato. Per raggiungere questo scopo vengono definite tre operazioni:

- Scritture stabili.** Una scrittura stabile consiste nello scrivere prima il blocco sul drive 1, e poi rileggerlo per verificare che sia stato scritto correttamente; se non è corretto riscrivere e rileggerlo per un massimo di n volte, finché l'operazione non viene completata con successo. Dopo n fallimenti consecutivi il blocco viene rimappato su di un blocco di riserva, e l'operazione è ripetuta fino ad ottenere un successo, non importa dopo quanti tentativi. Al completamento della scrittura sul drive 1, il blocco corrispondente sul drive 2 viene scritto e riletto, ripetutamente se è il caso, fino a che l'operazione non viene completata con successo. In assenza di crash di CPU, quando una scrittura stabile viene completata, il blocco è stato scritto correttamente su entrambi i dischi e verificato su entrambi.
- Letture stabili.** Una lettura stabile inizia leggendo un blocco dal drive 1; se questo porta un ECC non corretto, si riprova a leggere, per n volte. Se tutte le letture effettuate portano ECC non corretti, allora la lettura viene tentata sul blocco del drive 2. Siccome una scrittura stabile terminata con successo fornisce due copie buone di un blocco, secondo la nostra ipotesi che la probabilità che le due copie dello stesso blocco si rovinino entrambe in un ragionevole intervallo di tempo è trascurabile, la lettura stabile viene sempre completata con successo.
- Ripristino da crash.** Dopo un crash, un programma di ripristino analizza entrambi i dischi, confrontando i blocchi corrispondenti. Se una coppia di blocchi corrispondenti sono entrambi buoni e uguali, si prosegue. Se uno di essi presenta un ECC errato, il blocco corrotto viene sovrascritto con il contenuto del blocco buono. Se entrambi i blocchi sono buoni ma diversi, il blocco del drive 1 è copiato sul blocco del drive 2.

In assenza di crash di CPU, questo schema funziona sempre in quanto le scritture stabili producono sempre due copie valide di ogni blocco e si suppone che non si verifichino mai errori spontanei su due blocchi corrispondenti allo stesso tempo. Cosa succede nel caso di crash durante una scrittura stabile? Dipende da quando avviene il crash: ci sono cinque possibilità, descritte in Figura 5.30.

In Figura 5.30(a), il crash avviene prima che entrambe le copie del blocco siano state scritte. Durante il ripristino non sarà cambiato nulla e i vecchi valori continueranno ad esistere, il che è permesso.

In Figura 5.30(b), il crash di CPU avviene durante la scrittura del drive 1 e distrugge il contenuto del blocco. Comunque il programma di ripristino rileva questo errore e ripristina il blocco sul drive 1 dal drive 2. Quindi l'effetto del crash è eliminato e lo stato precedente viene ripristinato completamente.

In Figura 5.30(c), il crash di CPU avviene dopo che il blocco è stato scritto sul drive 1, ma non sul drive 2. Il punto di non ritorno viene passato a questo punto; il programma

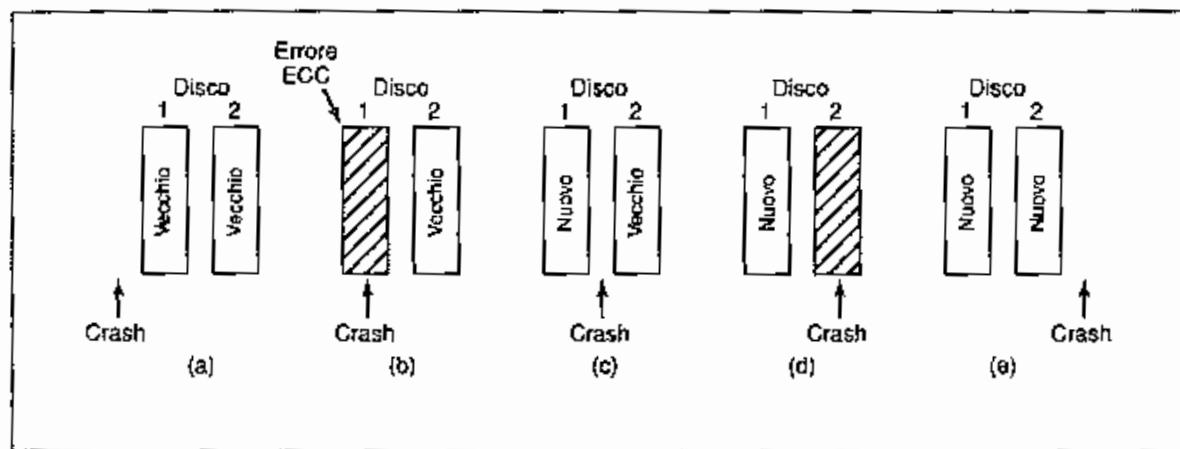


Figura 5.30 Analisi dell'influenza di un crash su una scrittura stabile.

di ripristino copia il blocco dal drive 1 al 2 e quindi la scrittura è stata completata con successo.

La Figura 5.30(d) è come la Figura 5.30(b): durante il ripristino, il blocco buono sovrascrive quello rovinato. Il valore finale di entrambi i blocchi è quello nuovo.

Infine, in Figura 5.30(e) il programma di ripristino vede che entrambi i blocchi sono uguali e quindi nulla viene cambiato e anche in questo caso la scrittura è completata con successo.

Sono possibili varie ottimizzazioni e miglioramenti di questo schema. Tanto per cominciare, confrontare tutte le coppie di blocchi dopo un crash è fattibile ma costoso; un grande miglioramento consiste nel tenere traccia di quale blocco sia stato scritto durante una scrittura stabile così da controllare solo un blocco durante il ripristino. Alcuni computer hanno una piccola quantità di **RAM non volatile**, che è una speciale memoria CMOS alimentata da una batteria al litio. Queste batterie durano per anni, forse per l'intera vita del computer. A differenza della memoria centrale, che viene persa dopo un crash, la RAM non volatile non viene persa. L'ora viene solitamente salvata qui (e incrementata da uno speciale circuito), ragione per cui i computer si ricordano l'ora anche dopo essere stati scollegati dalla corrente.

Supponendo che alcuni byte di RAM non volatile siano disponibili ad uso del sistema operativo, la scrittura stabile può inserire il numero del blocco che sta per modificare nella RAM non volatile, prima di iniziare la scrittura. Dopo il completamento della scrittura con successo, il numero di blocco nella RAM non volatile viene sovrascritto con un numero di blocco non valido, ad esempio -1. In queste circostanze, dopo un crash, il programma di ripristino può controllare la memoria non volatile per vedere se al momento del crash c'era una scrittura stabile in corso e, in questa eventualità, capire quale blocco può essere stato coinvolto. Le due copie del blocco vengono quindi controllate, in termini di correttezza e consistenza.

Se la RAM volatile non è disponibile, può essere simulata nel modo seguente. All'inizio della scrittura stabile un blocco fisso del disco nel drive 1 viene sovrascritto con il numero del blocco che sta per essere modificato; questo blocco viene riletto e verificato, finché la scrittura non è corretta; a questo punto viene scritto il blocco corrispondente sul drive 2. Quando la scrittura stabile è completata, entrambi i blocchi sono sovrascritti con un numero di blocco non valido e verificati. Anche in questo caso, dopo un crash, è facile capire quale scrittura stabile era in corso al momento del crash; naturalmente questa tecnica richiede ulteriori operazioni di disco, quindi dovrebbe essere usata con moderazione.

Occorre tenere in considerazione un ultimo punto. Abbiamo supposto che, ogni giorno, possa verificarsi solo un decadimento spontaneo di un blocco buono in una coppia di blocchi. Nei giorni successivi, però, anche l'altro blocco potrebbe rovinarsi, quindi una volta al giorno si deve fare un controllo completo di entrambi i dischi per riparare eventuali guasti. In questo modo, ogni mattina entrambi i dischi sono identici. Anche se entrambi i blocchi in una coppia vengono danneggiati a distanza di pochi giorni, tutti gli errori che si verificano possono venire riparati.

5.5 I clock

I clock (chiamati anche **timer**) sono, per vari motivi, essenziali nelle operazioni di un qualunque sistema multiprogrammato. Fra le altre cose, mantengono la data e l'ora, e impediscono che un processo monopolizzi il processore. Il software del clock prende spesso la forma di driver di dispositivo, sebbene il clock non sia né un dispositivo a blocchi, come il disco, né a carattere, come il mouse. La nostra analisi dei clock seguirà lo stesso schema dei paragrafi precedenti: prima di tutto daremo un'occhiata all'hardware dei clock, e in seguito al software dei clock.

5.5.1 L'hardware dei clock

Nei computer si usano comunemente due tipi di clock, entrambi abbastanza diversi dai cronometri e dagli orologi usati dalla gente comune, quello più semplice è legato alla linea di alimentazione a 110 o a 220 volt, e genera un'interruzione ogni ciclo del voltaggio, a 50 o a 60 Hz. Un tempo questi clock erano molto diffusi, ma oggi sono rari.

L'altro tipo è costituito da tre componenti: un oscillatore a cristallo, un contatore e un registro di caricamento, come in Figura 5.31. Quando un pezzetto di cristallo di quarzo viene tagliato in maniera opportuna, e messo sotto tensione, può essere forzato a generare un segnale periodico di alta precisione, tipicamente nell'ordine delle centinaia di MHz, a seconda del cristallo scelto. Con una opportuna elettronica, questo segnale base può essere moltiplicato per un piccolo fattore intero, per ottenere frequenze fino a 1000 MHz o anche di più. Almeno uno di questi circuiti si trova su ogni computer, e fornisce un segnale di sincronizzazione ai vari circuiti del computer. Questo segnale viene man-

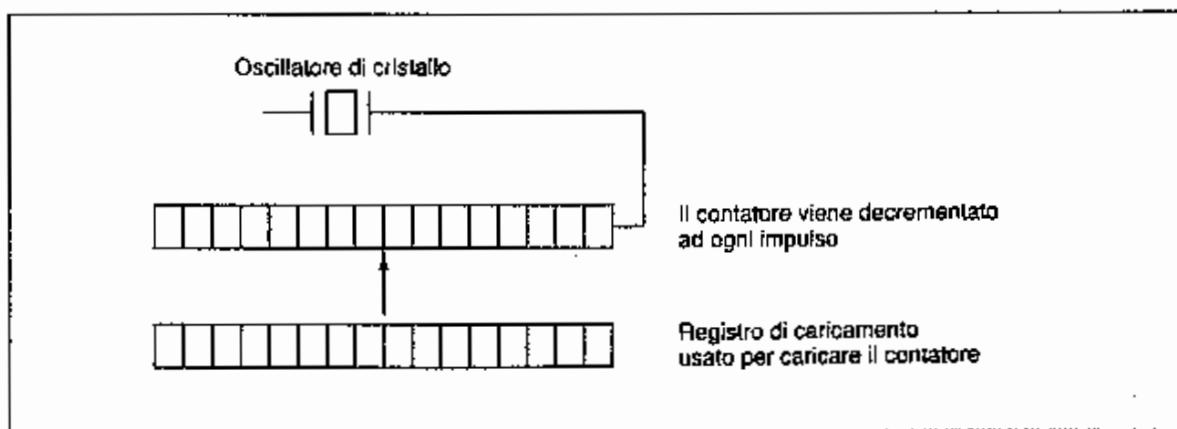


Figura 5.31 Un clock programmabile

dato al contatore per far sì che esso conti all'indietro fino allo zero. Quando il contatore raggiunge lo zero, causa un'interruzione di CPU.

I clock programmabili presentano diversi modi di funzionamento caratteristici. Nel modo a colpo singolo (one-shot-mode), quando viene fatto partire, il clock copia il valore del registro di caricamento nel contatore, e poi lo decrementa ogni volta che arriva un impulso dal cristallo. Quando il contatore arriva a zero, esso causa un'interruzione e si ferma finché non viene esplicitamente fatto ripartire dal software. Nel modo ad onda quadra (square-wave-mode), dopo essere arrivato allo zero ed aver causato l'interruzione, il registro di caricamento viene copiato automaticamente nel registro contatore, e tutto il processo si ripete all'infinito, queste interruzioni periodiche vengono chiamate tic (battiti del clock).

Il vantaggio del clock programmabile consiste nel fatto che la frequenza delle interruzioni può essere controllata dal software. Se si usa un quarzo da 500 MHz, il contatore verrà decrementato ogni 2 nanosecondi, con un registro contatore da 32 bit senza segno, le interruzioni si potrebbero programmare perché avvengano da una volta ogni 2 nanosecondi a una volta ogni 8,6 secondi. I clock programmabili su circuito integrato contengono di solito due o tre clock programmabili indipendentemente, ed hanno anche molte altre funzionalità (per esempio, la possibilità di contare verso l'alto anziché verso lo zero, la disabilitazione delle interruzioni, eccetera).

Per realizzare un clock che mantenga la data e l'ora quando il computer viene spento, la maggior parte dei computer ha un clock di backup alimentato a batterie, implementato con il tipo di circuiti a bassa potenza usati per gli orologi digitali. Il clock di backup può essere letto all'avvio; se il clock di backup non è presente, il software può chiedere all'utente la data e l'ora corrente. Esiste inoltre un modo standard con cui un sistema in rete può procurarsi l'ora corrente da un host remoto, in ogni modo l'ora e la data corrente vengono in seguito tradotte nel numero dei tic trascorsi dalle ore 12 UTC (Universal Coordinated Time, ora universalmente coordinata, un tempo nota come GMT o ora di Greenwich) dal primo gennaio 1970, nel caso di UNIX, o da un altro punto di riferimento: il tempo di inizio per Windows è il primo di gennaio 1980, e ad ogni tic, poi, il tempo reale viene incrementato di un'unità. Solitamente esistono programmi di utilità che permettono di impostare manualmente il clock di sistema e il clock di backup, e di sincronizzare i due.

5.5.2 Il software dei clock

Tutto ciò che fa l'hardware del clock, è generare un'interruzione ad istanti di tempo ben definiti, ogni altra cosa che riguarda il tempo deve essere realizzata dal software, tramite il driver del clock. I compiti esatti del driver di clock variano da sistema a sistema, ma generalmente essi includono la maggior parte delle seguenti funzioni:

1. Mantenere la data e l'ora.
2. Evitare che un processo resti attivo più a lungo di quanto dovrebbe.
3. Tenere una contabilità dell'uso del processore (CPU accounting).
4. Gestire le chiamate di sistema di tipo alarm da parte dei processi utente.
5. Mettere a disposizione dei timer di controllo per parti del sistema operativo.
6. Eseguire il profiling, il monitoraggio e la raccolta di statistiche.

La prima funzione del clock, la manutenzione della data e dell'ora attuale (detto anche tempo reale), non presenta alcuna difficoltà. Essa richiede semplicemente di incrementa-

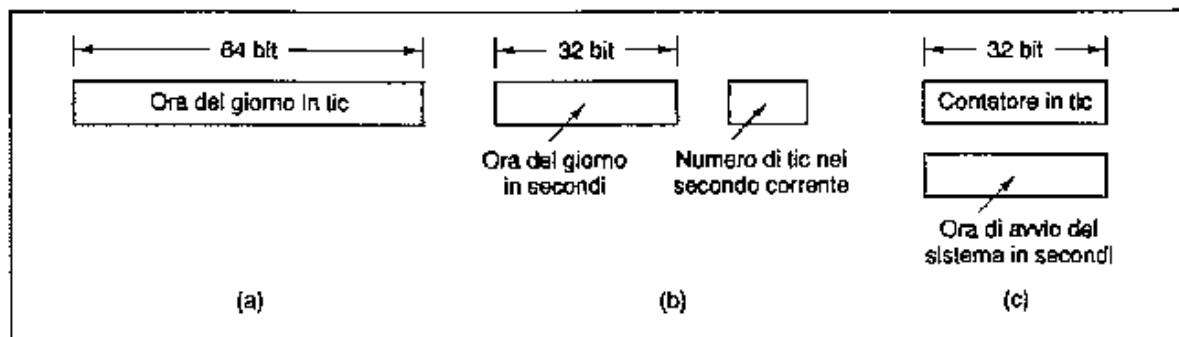


Figura 5.32 Tre modi per mantenere il tempo reale

re un contatore ad ogni tic, come detto precedentemente: con una frequenza di tic di 60 Hz, un contatore da 32 bit va in overflow dopo circa due anni, e quindi il sistema non può memorizzare il tempo reale a partire dal 1 gennaio 1970 in un registro a 32 bit.

Per risolvere questo problema si possono applicare tre diverse soluzioni. La prima è quella di usare un contatore a 64 bit, sebbene ciò renda più pesante l'operazione di incremento, poiché va fatta molte volte in un secondo. La seconda soluzione è quella di mantenere il tempo reale in secondi, anziché in tic, usando un contatore ausiliario per contare i tic fino allo scadere del secondo. Poiché 2^{32} secondi sono più di 136 anni, questo metodo funzionerà bene fino al ventiduesimo secolo.

Nella terza soluzione, si mantengono ancora i tic ma, anziché contarli relativamente a un qualche evento esterno, essi vengono contati a partire dal momento in cui è stato fatto partire per l'ultima volta il sistema (tempo di boot). Quando si legge il clock di backup, oppure l'utente digita dalla tastiera il tempo reale, si calcola il tempo di boot del sistema a partire dal tempo reale attuale, e lo si memorizza poi in una forma conveniente, più tardi, quando verrà richiesto il tempo reale attuale, il tempo reale memorizzato verrà aggiunto al contatore per ottenere la data e l'ora. Tutte e tre queste soluzioni sono mostrate nella Figura 5.32.

La seconda funzione del clock è impedire che un processo giri per un tempo maggiore del dovuto. Quando si fa partire un processo, lo scheduler inizializza un contatore con un valore corrispondente al quanto di tempo (in tic) che gli è stato assegnato; ad ogni interruzione del clock, il driver decrementa questo contatore di una unità. Quando arriva a zero, il driver richiama lo scheduler, per far sì che venga mandato in esecuzione un altro processo.

La terza funzione del clock è la contabilità del processore. La maniera più precisa per svolgere questo compito, è quella di far partire un secondo timer, diverso dal timer principale del sistema, ogni volta che un processo viene mandato in esecuzione; quando il processo viene fermato, si può leggere questo timer per sapere per quanto tempo ha girato. Per fare le cose con precisione, il secondo timer dovrebbe essere salvato quando avviene un'interruzione, e ripristinato in seguito.

Una maniera di procedere più semplice, ma meno accurata, è quella di mantenere un puntatore all'elemento corrispondente al processo corrente nella tabella dei processi, in una variabile globale; ad ogni tic, si incrementa poi un campo dell'elemento relativo al processo corrente in questa tabella; in questo modo, ogni tic viene "addebitato" al processo che sta girando in quel momento. Un problema secondario di questa strategia è che, se mentre un processo gira si verificano molte interruzioni, ad esso verrà comunque messo in conto l'intero tic, anche se ha fatto molto poco lavoro. Ad ogni modo, tenere conto delle interruzioni nella contabilità del processore è troppo costoso, e non si fa mai.

In molti sistemi, un processo può richiedere al sistema operativo di inviargli un avvi-

so dopo un certo intervallo di tempo, costituito di solito da un segnale, un'interruzione, un messaggio o qualcosa di simile. Un'applicazione che richiede simili avvisi è il software di rete, in cui se non si riceve l'acknowledgement di un pacchetto entro un certo intervallo di tempo, lo si trasmette nuovamente. Un'altra applicazione è l'istruzione assistita dal calcolatore, in cui si comunica la soluzione a uno studente che non abbia risposto alle domande entro il tempo previsto.

Se il driver avesse abbastanza clock, potrebbe adoperarne uno diverso per ogni singola richiesta; siccome ciò normalmente non avviene, esso deve simulare clock virtuali multipli mediante un singolo clock reale. Una soluzione è quella di mantenere una tabella, nella quale si memorizzano i tempi dei segnali per tutti i timer dipendenti da quello reale, e di mantenere anche in una variabile il tempo al quale deve essere inviato il prossimo segnale. Ogni volta che si aggiorna il tempo reale, il driver controlla se è arrivata l'ora del segnale più recente, in questo caso si ricerca nella tabella il tempo del prossimo segnale.

Se si aspettano molti segnali, è più efficiente simulare clock multipli concatenando tutte le richieste pendenti, ordinate rispetto al tempo, in un'unica lista, come mostrato dalla Figura 5.33. Ciascun elemento della lista indica quanti tic bisogna aspettare dopo il segnale che lo precede per inviarne un altro. In questo esempio, ci sono richieste di segnale pendenti per i tempi 4203, 4207, 4213, 4215 e 4216.

Nel caso della Figura 5.33, la prossima interruzione arriverà fra tre tic; ad ogni tic viene decrementato *Prossimo Segnale*; quando questo arriva a zero, viene inviato il segnale relativo al primo elemento della lista, e si cancella questo elemento. Quindi, in *Prossimo Segnale* viene messo il valore contenuto dall'elemento che si trova attualmente in testa alla lista, cioè, in questo esempio, 4.

Si noti che, durante un'interruzione del clock, il driver deve fare diverse cose: incrementare il tempo reale, decrementare il quanto di tempo e controllare se è arrivato a zero, contabilizzare il tempo di CPU da addebitare, e decrementare il contatore d'allarme. Ad ogni modo, ognuna di queste operazioni è stata accuratamente studiata per essere molto veloce, visto che tutte devono essere ripetute molte volte ogni secondo.

Alcune parti del sistema richiedono l'inizializzazione di timer, che sono chiamati *watchdog timer* (timer di controllo). Per esempio, i floppy disk non ruotano quando non sono usati, per evitare problemi di usura del supporto e della testina. Per usare un floppy disk, il sistema deve far partire il motore del disco, e solo quando il floppy disk sta ruotando alla massima velocità si può iniziare l'I/O. Quando un processo cerca di leggere da un floppy disk inattivo, il driver del floppy disk fa partire il motore ed imposta il watchdog timer in modo che causi un'interruzione dopo un periodo di tempo sufficientemente lungo (in quanto non esiste un'interruzione da parte del floppy disk stesso, che indica quando il disco ha raggiunto la velocità giusta).

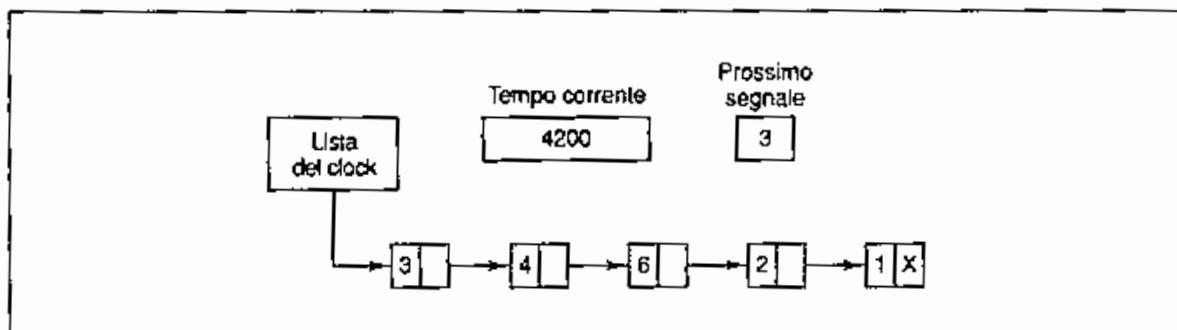


Figura 5.33 Simulare più timer con un clock solo

Il meccanismo usato dal driver del clock per trattare i watchdog timer è lo stesso di quello usato per i segnali utente; la sola differenza è che, quando un timer arriva a zero, anziché provocare un segnale, il driver chiama una procedura fornita dal chiamante. Questa procedura fa parte del codice del chiamante, e può fare qualsiasi cosa ritenga necessaria, anche generare un'interruzione, sebbene a livello del nucleo le interruzioni non siano convenienti e i segnali non esistano. Questa è proprio la ragione per cui si mettono a disposizione i watchdog timer. Si noti che il meccanismo dei watchdog funziona solo quando il driver del clock e la procedura da chiamare condividono lo stesso spazio di indirizzamento.

L'ultimo punto della nostra lista riguarda l'attività di profiling (rilevamento del profilo d'uso): alcuni sistemi operativi mettono a disposizione un meccanismo con il quale un processo utente può chiedere al sistema di costruire un istogramma (profile) del suo contatore di istruzioni, permettendo di controllare in quali parti di codice il processo passa il proprio tempo. Quando l'attività di profiling è ammessa, ad ogni tic il driver del clock controlla se il processo attualmente in esecuzione è sotto profiling e, in questo caso, calcola l'intervallo di indirizzi corrispondente al contatore di istruzioni attuale. In seguito, incrementa il contatore dell'intervallo corrispondente di 1; questo sistema può anche essere usato per fare il profiling del sistema.

5.5.3 Timer soft

La maggior parte dei computer ha un secondo clock programmabile che può essere impostato in modo da causare delle interruzioni di temporizzazione a qualunque frequenza richiesta da un programma. Questo timer è in aggiunta al timer centrale di sistema le cui funzioni sono state appena descritte. Finché la frequenza di interruzione è bassa, non ci sono problemi ad usare questo secondo timer per scopi specifici di un'applicazione; il problema si pone se la frequenza del timer specifico dell'applicazione è molto alta. Nel seguito, descriviamo brevemente uno schema di timer software che funziona bene in molte circostanze, anche ad alte frequenze. L'idea è dovuta ad Aron e Druschel (1999), per maggiori dettagli, riferitevi al loro articolo.

Generalmente ci sono due modi di gestire l'I/O: tramite interruzioni e tramite meccanismi di polling. Le interruzioni hanno una latenza bassa, ossia si verificano immediatamente dopo l'evento stesso senza alcun ritardo o con un ritardo molto piccolo. D'altro canto, con le CPU moderne, le interruzioni hanno un overhead sostanziale, dovuto al bisogno di fare cambi di contesto ed alla loro influenza su pipeline, TLB e cache.

L'alternativa alle interruzioni è che l'applicazione stessa si ponga in attesa dell'evento, evitando le interruzioni. Però si può avere un certo tempo di latenza, in quanto un evento può verificarsi subito dopo il polling, nel qual caso deve aspettare quasi per l'intero intervallo di polling. In media, la latenza è della metà dell'intervallo di polling.

Per alcune applicazioni non sono accettabili né l'overhead delle interruzioni né la latenza del polling. Si consideri ad esempio una rete ad alte prestazioni quali la Gigabit Ethernet. Questa rete è in grado di ricevere o accettare un pacchetto intero ogni 12 μ s; per ottenere delle prestazioni ottimali, si dovrebbe spedire un pacchetto ogni 12 μ s.

Un modo per ottenere questa frequenza è che al completamento della trasmissione di un pacchetto si verifichi un'interruzione o che il secondo timer causi un'interruzione ogni 12 μ s. Il problema è che sono state fatte delle misurazioni che provano che tale interruzione richiede 4,45 μ s su un Pentium II a 300 MHz (Aron e Druschel, 1999). Questo overhead non è molto migliore di quello di un computer degli anni settanta: sulla

maggior parte dei minicomputer, per esempio, un'interruzione impiegava quattro cicli di bus, per inserire il PC e il PSW sulla pila e caricare il PC nuovo e il nuovo PSW. Oggi-giorno, con la pipeline, la MMU, la TLB e la cache, si ha molto più overhead. Questi effetti tendono a peggiorare piuttosto che migliorare col tempo, eliminando i vantaggi di avere frequenze di clock più alte.

I timer soft eliminano le interruzioni: mentre il kernel è in esecuzione per un qualsiasi motivo, prima di ritornare in modo utente, controlla il clock real-time per vedere se il timer soft è scaduto; in questo caso, esegue l'evento programmato (ad esempio la trasmissione di un pacchetto o il controllo di eventuali pacchetti in ingresso), senza bisogno di passare in modo kernel dato che il sistema è già in tale modo. Alla fine dell'esecuzione, il timer software viene reimpostato per una scadenza successiva: quello che si fa è copiare il valore di clock corrente sul timer e aggiungere ad esso l'intervallo dopo il quale lo si vuole attivare.

I timer software funzionano o meno a seconda della frequenza in cui il kernel viene usato per altri motivi. Questi motivi sono:

1. Chiamate di sistema.
2. Accessi mancati alla TLB.
3. Fault di pagina.
4. Interruzioni di I/O.
5. CPU che si mette in attesa.

Per capire quanto spesso si verificano questi eventi, Aron e Druschel effettuarono diverse misurazioni con diversi carichi di CPU, tra cui un server Web completamente carico, un server Web con un job in background pesante dal punto di vista computazionale, suonare audio real-time direttamente da Internet, e ricompilare un kernel UNIX. La frequenza media di ingresso nel kernel variava da 2 μ s a 18 μ s e di questi accessi la metà sono dovuti a chiamate di sistema. Pertanto, in prima approssimazione, è possibile avere un timer soft che si attiva ogni 12 μ s, anche se talvolta può mancare un scadenza. Per applicazioni quali l'invio di pacchetti o l'acquisizione di pacchetti in ingresso, un occasionale ritardo di 10 μ s è meglio di un'interruzione che si mangia il 35% della CPU.

Naturalmente, ci saranno periodi privi di chiamate di sistema, di accessi alla TLB o fault di pagina, nel qual caso il timer soft non verrà attivato. Se vogliamo dare un limite massimo a questi intervalli, possiamo impostare un secondo timer hardware in modo che si attivi, ad esempio, ogni millisecondo. Se l'applicazione può sopravvivere con solo 1000 pacchetti al secondo per certi intervalli di tempo, allora la combinazione di timer soft e timer hardware a frequenza bassa, può essere meglio di un I/O puramente guidato da interruzioni o puro polling.

5.6 I terminali orientati a carattere

Ogni calcolatore dispone di almeno una tastiera e un display (monitor o schermo piatto) usati per comunicare con esso. Anche se la tastiera e il display in un PC sono tecnicamente dispositivi separati, essi lavorano a stretto contatto; nel caso di mainframe, ci sono solitamente molti utenti remoti ognuno dei quali è dotato di un dispositivo contenente una tastiera e un display: questi dispositivi sono stati chiamati storicamente **terminali**. Continueremo ad usare questo termine, anche parlando di personal computer (principalmente in mancanza di un termine migliore).

Esistono diversi tipi di terminale. Tra i tipi che si incontrano più comunemente oggi-giorno, abbiamo:

1. Terminali "standalone" con interfacce seriali RS-232 per l'uso con mainframe.
2. Display per personal computer dotati di interfaccia utente grafica.
3. Terminali di rete.

Ognuno di questi ha la propria nicchia ecologica; nei prossimi paragrafi li descriveremo ad uno ad uno.

5.6.1 L'hardware dei terminali RS-232

I terminali RS-232 sono dispositivi che dispongono di una tastiera e di uno schermo, e che comunicano usando un'interfaccia seriale, un bit alla volta (vedi Figura 5.34). Questi terminali usano connettori a 9 o a 25 piedini (pin), dei quali uno viene usato per trasmettere i dati, uno per riceverli e uno per la massa; gli altri piedini sono dedicati a varie funzioni di controllo, la maggior parte delle quali non viene generalmente usata. Le linee su cui i caratteri vengono spediti un bit alla volta (e non con 8 bit in parallelo, nel modo in cui le stampanti sono interfacciate ai PC) vengono dette **linee seriali**; anche tutti i modem usano questa interfaccia. In UNIX, le linee seriali hanno nomi quali */dev/tty1* e */dev/tty2*, mentre in Windows nomi come *COM1* e *COM2*.

Per spedire un carattere lungo una linea seriale ad un terminale RS-232 o a un modem, il calcolatore deve trasmetterlo un bit alla volta, preceduto da un bit di start e seguito da uno o due bit di stop che delimitano il carattere; un bit di parità, che fornisce un metodo di rilevazione degli errori rudimentale, può essere inserito a precedere i bit di stop, ma questo è obbligatorio solo per la comunicazione con sistemi mainframe.

I terminali RS-232 sono ancora usati tipicamente nel mondo dei mainframe, per permettere all'utente remoto di comunicare col mainframe, talvolta usando un modem e la linea telefonica; sono diffusi nei mondi delle compagnie aeree, nelle banche, e in altre industrie. Anche quando vengono sostituiti da personal computer, il PC spesso emula i vecchi terminali RS-232, in modo da evitare di sostituire il software del mainframe.

Questi terminali un tempo dominavano il mondo dei minicomputer; una grande quantità di software per sistemi che sono cresciuti in quegli anni si basano su questi terminali; tutti i sistemi UNIX, per esempio, supportano questo tipo di dispositivo.

Comunque, è più importante che molti sistemi UNIX attuali (e molti altri sistemi) dan-

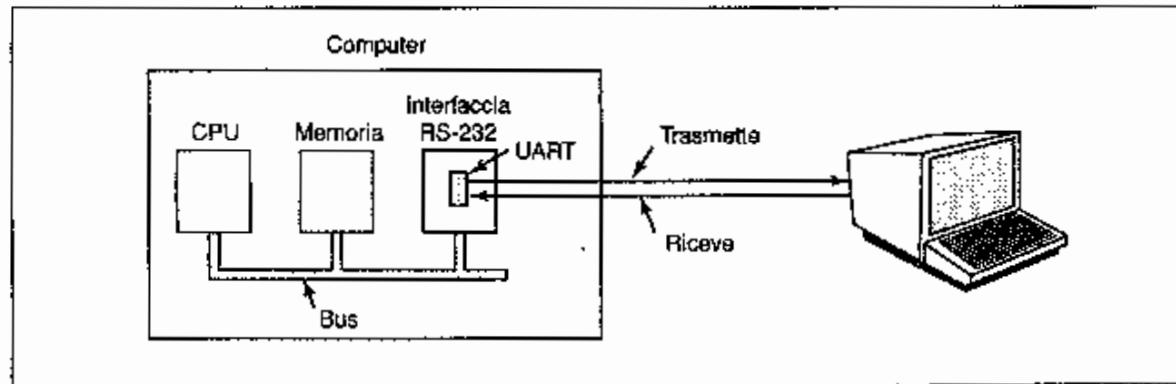


Figura 5.34 Un terminale RS-232 che comunica con un computer lungo una linea di comunicazione, un bit alla volta.

no modo di creare una finestra formata da un certo numero di linee di testo: molti programmati lavorano quasi esclusivamente in modo testo all'interno di tali finestre, anche su personal computer o su stazioni di lavoro. Queste finestre solitamente emulano terminali RS-232 (o lo standard ANSI per questo tipo di terminali) in modo da poter eseguire la grande quantità di software esistente che era stato sviluppato per tali terminali. Nel corso degli anni questo software, quali gli editor *vi* e *emacs*, è diventato ben collaudato ed estremamente stabile, e quindi molto apprezzato dai programmati.

Il software di tastiera e terminale per queste finestre di emulazione terminale è lo stesso che per i veri terminali. Siccome gli emulatori sono molto usati, il software è ancora molto importante, e quindi lo descriveremo nelle prossime sezioni.

I terminali RS-232 sono orientati a carattere, questo significa che lo schermo o la finestra mostrano un certo numero di linee di testo, ognuna con una dimensione massima; una dimensione tipica è 25 linee di 80 caratteri l'una. Mentre solitamente sono supportati alcuni caratteri speciali, questi terminali (e i loro emulatori) funzionano praticamente a solo testo.

Poiché sia i calcolatori sia i terminali lavorano internamente a caratteri, ma comunicano fra di loro in maniera seriale, un bit alla volta, sono stati sviluppati dei circuiti integrati che svolgono le funzioni di conversione da carattere a seriale e viceversa. Essi vengono chiamati **UART** (*ricevitori trasmettitori universali e asincroni*, Universal Asynchronous Receiver Transmitters); gli UART vengono connessi al calcolatore inserendo una scheda di interfaccia RS-232 nel bus, come mostrato dalla Figura 5.34. Su molti computer, una o due porte seriali sono inserite nella scheda madre.

Per visualizzare un carattere, il driver del terminale lo scrive sulla scheda di interfaccia, dove viene memorizzato e poi, mediante un'operazione di shift realizzata dall'UART, spedito un bit alla volta sulla linea seriale. Per esempio, per un modem analogico che opera a 56000 bps, per spedire un carattere occorrono poco più di 179 μ s. Come risultato di questa bassa velocità di trasmissione, il driver del terminale spedirà, in generale, un carattere alla volta verso la scheda di interfaccia RS-232, e poi si fermerà in attesa dell'interruzione che segnala che il carattere è stato spedito, e che la UART è pronta a riceverne un altro. L'UART può spedire e ricevere caratteri simultaneamente; viene generata un'interruzione anche quando viene ricevuto un carattere, e solitamente può essere mantenuto nel buffer un numero limitato di caratteri in input. Il driver del terminale deve controllare un registro nel momento in cui riceve un'interruzione, per determinare la causa dell'interruzione; ma alcune schede di interfaccia sono dotate di un proprio processore e di una propria memoria, e possono gestire più di una linea seriale alla volta, alleggerendo il processore centrale di gran parte del carico relativo all'I/O.

I terminali RS-232 possono essere divisi in tre categorie. I più semplici sono i terminali stampanti (di *hardcopy*), che spediscono al calcolatore i caratteri che vengono battuti sulla tastiera, e stampano su carta quelli che ricevono dal calcolatore. Questi terminali sono obsoleti e oramai si incontrano raramente, tranne che come stampanti di fascia bassa.

I terminali video CRT "stupidi" funzionano nella stessa maniera, con la differenza che, anziché scrivere sulla carta, scrivono su uno schermo a raggi catodici. Essi vengono spesso chiamati "tty di vetro" perché il loro funzionamento è identico ai tty di *hardcopy*. (Il termine "tty" è un'abbreviazione di Teletype⁴, una delle ditte pioniere nel campo dei terminali per calcolatori; "tty" è diventato un termine comune per indicare genericamente un terminale). Anche i tty di vetro sono obsoleti.

I terminali video CRT "intelligenti" sono di fatto piccoli calcolatori specializzati. Essi dispongono di un processore e di memoria, e contengono programmi complessi, di solito memorizzati su ROM. Dal punto di vista del sistema operativo, la differenza principale fra un terminale video stupido ed uno intelligente sta nel fatto che quest'ultimo interpreta alcune sequenze di caratteri di controllo. Per esempio, spedendo il carattere ESC

(codice ASCII 033, 0x1B) seguito da vari altri caratteri, è possibile spostare il cursore in una posizione qualunque dello schermo, inserire un testo in mezzo allo schermo, e così via. I terminali intelligenti sono quelli usati nei sistemi mainframe e sono quelli emulati dagli altri sistemi operativi; nel seguito parleremo del loro software.

5.6.2 Il software di input

La tastiera e lo schermo sono dispositivi praticamente indipendenti e quindi verranno trattati separatamente. In effetti, essi non sono completamente indipendenti, in quanto i caratteri che vengono battuti sulla tastiera devono di solito essere mostrati sullo schermo.

Il compito principale del driver della tastiera, è quello di raccogliere i caratteri battuti, e di distribuirli ai programmi utente quando questi chiamano la procedura di lettura dal terminale. Per questo tipo di driver si possono seguire due tipi di filosofie: nella prima, il compito del driver è semplicemente quello di raccogliere i dati in ingresso e di passarli, senza modifiche, ai livelli superiori. In questo caso, un programma che legge dal terminale riceve una sequenza di codici ASCII. (Passare al programma utente i numeri dei tasti è una soluzione troppo grossolana, e rende i programmi molto dipendenti dalla macchina).

Questa filosofia si sposa molto bene con le necessità di editor di schermo molto sofisticati, come per esempio *emacs*, che permette all'utente di associare ad un tasto, o ad una sequenza di tasti, una azione qualsiasi. Significa comunque che, se l'utente batte *dste* anziché *date* e poi corregge l'errore battendo tre caratteri di cancellazione e poi *ate* seguito da un line feed, al programma utente saranno passati tutti gli 11 codici ASCII relativi a questi caratteri, nel seguente modo:

```
d s t e ←←← a t e CR.
```

La maggior parte dei programmi non ha bisogno di un simile livello di dettaglio; spesso hanno bisogno solo dei dati in ingresso già corretti e non della sequenza di tasti che li ha prodotti. Questa osservazione porta alla seconda filosofia: il driver tratta l'*editing* di linea, e passa solo le linee già corrette ai programmi dell'utente. La prima filosofia è orientata al carattere, mentre la seconda è orientata alla linea; esse in origine venivano chiamate, rispettivamente, **modo raw** (crudo) e **modo cooked** (cucinato). Lo standard POSIX usa il termine meno pittoresco di **modo canonico** per descrivere il modo linea, mentre il **modo non canonico** è equivalente al modo raw, anche se permette di cambiare molti dettagli del comportamento del terminale. Sistemi compatibili con lo standard POSIX forniscono funzioni di libreria che permettono di selezionare uno dei due modi, e permettono di cambiare diversi aspetti nella configurazione del terminale.

Il primo compito di un driver della tastiera è quello di raccogliere i caratteri battuti; se ogni pressione di tasto genera un'interruzione, il driver può acquisire il codice del carattere battuto durante l'interruzione. Se le interruzioni sono trasformate in messaggi dal software di basso livello, è possibile inviare il codice del carattere appena battuto nel messaggio; in alternativa, si può inserire il carattere in un piccolo buffer in memoria centrale, e poi notificare al driver, tramite il messaggio, che è arrivato un carattere. L'ultima soluzione è più sicura se il messaggio può essere inviato solo ad un processo in attesa e se c'è qualche possibilità che il driver della tastiera possa ancora essere occupato a trattare il carattere precedente.

Se il terminale lavora in modo canonico (cooked), i caratteri devono essere memorizzati finché non è stata battuta l'intera linea, in quanto l'utente può decidere, in seguito, di cancellarne una parte. Anche nel caso di operazioni in modo raw, se il programma uten-

te non ha ancora richiesto la lettura, i caratteri vanno memorizzati per permettere di anticipare la battitura dei caratteri. (I progettisti di sistemi che non prevedono la possibilità per l'utente di anticipare la battitura dei caratteri dovrebbero essere ricoperti di catrame e poi di piume o, ancora peggio, essere costretti ad usare il sistema da loro progettato.)

Di solito, per la memorizzazione temporanea dei caratteri battuti da terminale, vengono adottate due soluzioni. Nella prima il driver contiene un insieme di buffer, ciascuno dei quali, per esempio, di 10 caratteri. Associata a ciascun terminale si trova una struttura dati che contiene, fra l'altro, un puntatore alla catena di buffer per i dati in ingresso raccolti per quel terminale. Via via che vengono battuti ulteriori caratteri, altri buffer vengono assegnati a quel terminale e collegati nella catena. Quando poi i caratteri sono passati al programma dell'utente, i buffer vengono rimossi e riassegnati all'insieme centrale.

L'altra soluzione consiste nel raccogliere i caratteri direttamente in un buffer della struttura dati del terminale, evitando l'uso di un insieme di buffer centrale. Poiché è frequente che gli utenti digitino un comando che richiede un certo tempo di esecuzione (per esempio una ricompilazione e un link di un grande programma binario), e poi continuano a battere alcune altre linee, per stare tranquillo il driver dovrà allocare qualcosa come 200 caratteri di buffer per ogni terminale. In un grosso sistema con un centinaio di terminali, questo significa riservare 20K per i buffer di tastiera, il che è chiaramente eccessivo visto che è probabilmente sufficiente un insieme di buffer centralizzato con spazio per circa 5K di caratteri. D'altra parte la presenza di un buffer dedicato per ogni terminale rende il driver più semplice (non c'è bisogno della gestione delle liste concatenate), e deve quindi essere preferita nel caso di personal computer con una sola tastiera. La Figura 5.35 mostra le differenze fra questi due metodi.

Sebbene la tastiera ed il video siano due dispositivi logicamente separati, molti utenti sono abituati a vedere comparire sullo schermo i caratteri che hanno appena battuto sulla tastiera. Alcuni terminali (più vecchi) realizzano questa funzione tramite meccanismi hardware, il che non solo costituisce una seccatura quando vengono immesse delle password, ma limita pesantemente la flessibilità degli editor sofisticati e di altri programmi. Per fortuna, la maggior parte dei terminali non visualizza nulla quando vengono battuti

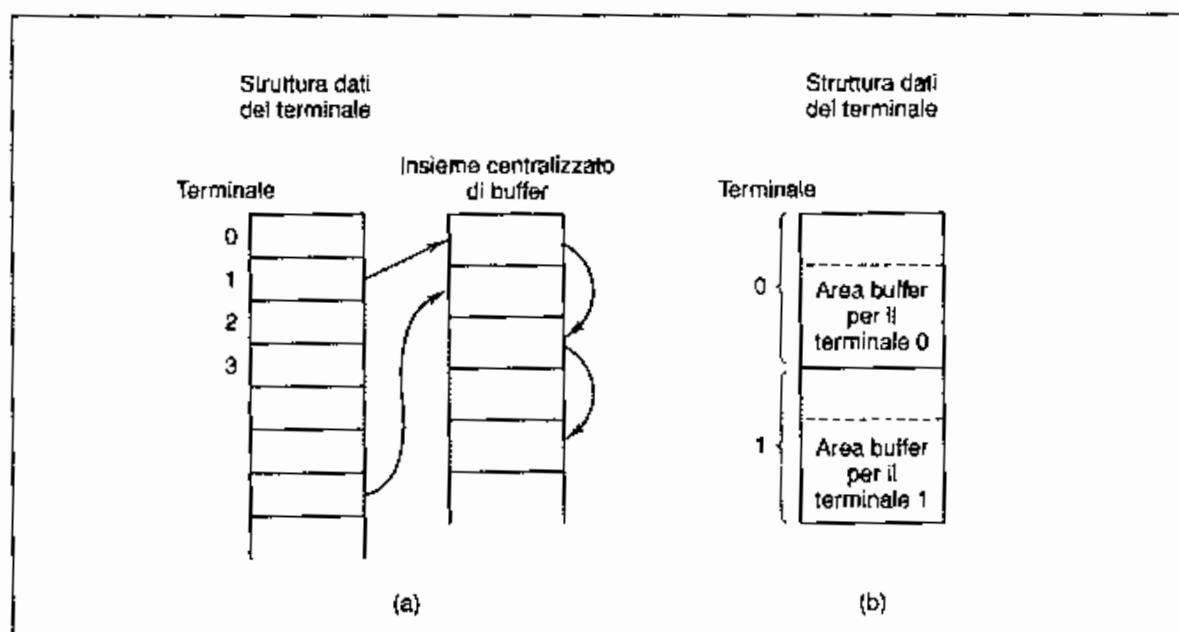


Figura 5.35 (a) Insieme centrale di buffer; (b) un buffer dedicato per ogni terminale.

dei caratteri. È compito del software riprodurre i dati in ingresso, se si desidera; questa operazione viene chiamata eco (echoing).

L'operazione di eco è resa complessa dal fatto che un programma può scrivere sul video mentre l'utente batte i caratteri sulla tastiera. Il driver della tastiera dovrà almeno prevedere dove scrivere i nuovi dati in ingresso, senza che essi vengano sovrascritti dai dati in uscita del programma.

L'operazione di eco risulta complicata anche quando vengono battuti più di 80 caratteri su un terminale che supporta 80 caratteri per riga, o un qualsiasi altro numero. A seconda dell'applicazione può essere giusto andare a capo quando la riga corrente è piena, ma qualche driver tronca semplicemente i dati in ingresso all'ottantesimo carattere, e butta via tutti gli altri.

Un altro problema si ha nella gestione dei caratteri di tabulazione (tab). Sta al driver calcolare la posizione corrente del cursore, tenendo in considerazione sia i dati in uscita dai programmi che quelli di eco, e calcolare il numero esatto di spazi da scrivere sul video.

Si esamini ora il problema dell'equivalenza dei dispositivi. Dal punto di vista logico, ci si aspetta che la fine di una linea sia contraddistinta da un ritorno carrello, che riporta il cursore sulla prima colonna, e da un carattere di line feed, che muove il cursore alla prossima riga. Non è proponibile richiedere che l'utente batte entrambi questi caratteri alla fine di ogni linea (sebbene alcuni terminali hanno un tasto che li spedisce entrambi, con una probabilità del 50% che essi vengano inviati nell'ordine che il software considera giusto). Sta ancora al driver convertire ciò che gli arriva dalla tastiera nella forma standard in cui il sistema operativo si aspetta questi dati.

Se la forma standard prevede di memorizzare la fine delle linee con un carattere di line feed (come nel caso di UNIX), allora tutti i ritorni carrello dovranno essere trasformati in line feed. Se la forma interna prevede entrambi i caratteri (come nel caso di Windows), il driver dovrà generare un ritorno carrello ogni volta che incontra un line feed e un line feed ogni volta che incontra un ritorno carrello. Indipendentemente dalla forma di memorizzazione interna, comunque, il terminale potrebbe richiedere sia il line feed sia il ritorno carrello, per far sì che lo schermo venga aggiornato correttamente. Poiché nei grossi sistemi si possono connettere terminali di tipo molto diverso, il driver della tastiera si deve occupare di tradurre nella forma standard del sistema operativo tutte le diver-

Carattere	Nome POSIX	Commento
CTRL-H	ERASE	Torna indietro di un carattere
CTRL-U	KILL	Cancella l'intera riga scritta
CTRL-V	LNEXT	Interpreta letteralmente il prossimo carattere
CTRL-S	STOP	Frena l'output
CTRL-Q	START	Inizia l'output
DEL	INTR	Processo di interruzione (SIGINT)
CTRL-\	QUIT	Forza core dump (SIGQUIT)
CTRL-D	EOF	Fine file
CTRL-M	CR	Ritorno carrello [non modificabile]
CTRL-J	NL	Fine linea [non modificabile]

Figura 5.36 Caratteri gestiti in modo speciale nel modo canonico.

se combinazioni di caratteri line feed e ritorno carrello, e di mandare in eco la sequenza di caratteri corretta.

In modo canonico, un certo numero di caratteri di ingresso ha un significato particolare. La Figura 5.36 mostra tutti i caratteri speciali richiesti da POSIX: sono di default tutti i caratteri di controllo che non entrano in conflitto con i testi in input o i codici usati dai programmi, ma tutti tranne gli ultimi due possono essere cambiati a scelta del programma.

Il carattere di cancellazione (ERASE) permette all'utente di cancellare il carattere appena battuto: questo carattere è solitamente il CTRL-H (backspace). Esso non viene aggiunto alla coda dei caratteri in ingresso, ma provoca la rimozione dalla coda dell'ultimo carattere battuto. Per rimuovere l'ultimo carattere battuto anche dallo schermo, bisogna eseguire un'operazione di eco di tre caratteri: un backspace, uno spazio e poi ancora un backspace. Se il carattere che deve essere cancellato fosse un tab, bisognerebbe tenere conto anche di come esso è stato gestito: se è stato espanso immediatamente in spazi, occorrono informazioni aggiuntive per capire fino a dove tornare indietro, se invece il tab stesso è stato memorizzato nella coda di input, lo si può rimuovere e rивisualizzare l'intera linea. Nella maggior parte dei sistemi, il carattere di cancellazione funziona solo all'interno di una riga, e non cancella il ritorno carrello né i caratteri che si trovano sulla riga precedente.

Quando l'utente rileva un errore all'inizio della riga che sta battendo, spesso è conveniente cancellare l'intera riga e ricominciare da capo: il carattere di KILL elimina l'intera linea. Molti sistemi fanno scomparire la linea eliminata dallo schermo, mentre altri la visualizzano seguita da un'andata a capo in quanto alcuni utenti vogliono vedere la linea vecchia, di conseguenza, effettuare un eco sul KILL è una questione di gusti. Così come nel caso dell'ERASE, solitamente non è possibile tornare più indietro della linea corrente; quando viene cancellato un blocco di caratteri, al driver può convenire rilasciare gli eventuali buffer usati all'insieme centralizzato, oppure no.

In alcuni casi, i caratteri di ERASE e di KILL devono essere passati come dati normali. Il carattere LNEXT serve come carattere di escape; in UNIX per difetto esso è CTRL-V. Ad esempio, molti dei vecchi sistemi UNIX usano il simbolo @ per KILL, ma il sistema di posta elettronica di Internet usa indirizzi nella forma *linda@cs.washington.edu*. Possono esserci persone che si sentono a proprio agio con le vecchie convenzioni e vogliono ridefinire il KILL come @, ma poi hanno bisogno di un @ letterale per gli indirizzi di e-mail; questo può essere fatto scrivendo CTRL-V @. Il CTRL-V stesso può essere scritto in modo letterale come CTRL-V CTRL-V. Dopo aver letto un CTRL-V, il driver imposta un flag per ricordare che il carattere successivo è esentato da elaborazione speciale; il carattere LNEXT stesso non viene inserito nella coda dei caratteri.

Per permettere all'utente di fermare un'immagine sullo schermo, vengono a volte messi a disposizione alcuni caratteri di controllo che permettono di bloccare lo scorrimento e di farlo ripartire. In UNIX questi caratteri sono, rispettivamente, lo STOP, (CTRL-S), e lo START, (CTRL-Q); essi non sono memorizzati, ma servono per mettere a 1 o a 0 un flag nella struttura dati del terminale. Quando viene richiesta una operazione di output sul terminale, si controlla il flag: se è a 1, non si produce alcuna uscita; di solito si sopprime l'operazione di eco, e l'output del programma.

Spesso è necessario eliminare un processo che, in fase di debugging, è sfuggito al controllo. Per questo scopo possono essere usati i tasti INTR (DEL) e QUIT (CTRL-\). In UNIX, il tasto DEL provoca l'invio di un segnale di SIGINT a tutti i processi che sono stati fatti partire da quel terminale. Realizzare correttamente questa funzione può non essere facile; la parte difficile è quella di inviare l'informazione dal driver del terminale alla parte del sistema operativo che gestisce i segnali, la quale, dopo tutto, non ha richiesto questa informazione. Il CTRL-\ è simile al DEL, ad eccezione del fatto che manda un segnale di SIGQUIT che, se non è accettato o ignorato, provoca un dump della memoria.

centrale sul disco. Quando viene battuto uno di questi due tasti, il driver deve fare l'echo di un line feed e di un ritorno carrello, e scartare tutti i caratteri correnti in ingresso per consentire di ripartire. Spesso il valore di default di INTR è CTRL-C invece di DEL, perché molti programmi usano DEL come cancellazione nell'editing, in modo intercambiabile col backspace.

Un altro carattere speciale è l'EOF (CTRL-D), che in UNIX fa sì che tutte le richieste correnti di input da terminale siano soddisfatte con i soli caratteri in ingresso attualmente disponibili, anche se il buffer è vuoto. Battere un CTRL-D all'inizio di una riga fa sì che il programma utente ritorni da una operazione di read con 0 caratteri letti, il che viene normalmente interpretato come fine del file, e provoca in molti programmi un comportamento simile a quello che si avrebbe se si facesse un'operazione di lettura su un file già arrivato alla fine.

Alcuni driver di terminali mettono a disposizione caratteristiche di editing di riga molto più sofisticate di quelle qui esposte; hanno caratteri speciali per cancellare parole, per andare avanti o indietro di un carattere o di una parola, per posizionarsi all'inizio o alla fine di una riga, inserire testo nel mezzo, e così via. Aggiungere tutte queste caratteristiche al driver del terminale lo rende molto più pesante e, per di più, esse si rivelano del tutto inutili quando si lavora con editor molto sofisticati che funzionano in ogni caso in modo raw.

5.6.3 Il software di output

L'output è più semplice dell'input, nella maggior parte dei casi il computer invia i caratteri al terminale ed è lì che essi vengono scritti. Solitamente, un blocco di caratteri, per esempio una linea, viene scritto sul terminale da una chiamata di sistema. Il metodo comunemente usato per i terminali RS-232 è di avere buffer di uscita associati ad ognuno di essi; i buffer possono provenire dallo stesso insieme di buffer di input oppure essere dedicati. Quando un programma scrive sul terminale, per prima cosa si copia l'uscita nei buffer, come avviene anche per i caratteri della funzione di eco. Dopo che tutti i caratteri sono stati copiati nei buffer, il primo carattere è mandato in uscita, e il driver va in stato di attesa, per ripartire quando termina l'interruzione e spedire il secondo carattere, ricominciando il ciclo.

Gli editori orientati al video e molti altri programmi abbastanza sofisticati hanno bisogno di aggiornare il video in una maniera complicata come ad esempio sostituendo una riga a metà schermo; per venire incontro a queste esigenze, molti driver di terminale mettono a disposizione sequenze di comandi per spostare il cursore, inserire e cancellare caratteri o linee in corrispondenza del cursore, eccetera: questi comandi sono spesso chiamati sequenze di escape. Ai tempi d'oro dei terminali RS-232, c'erano centinaia di tipi di terminali, ognuno con le proprie sequenze di escape e, come conseguenza, era difficile scrivere del software che funzionasse su più di un tipo di terminale.

Una soluzione, introdotta nel Berkeley UNIX, era un database di terminali chiamato *termcap*. Questo pacchetto software definiva un certo numero di azioni base, quali quelle di muovere il cursore nella posizione (riga, colonna); per muovere il cursore in una posizione specifica, il software, ad esempio l'editor, usava una sequenza di escape generica, che poi veniva convertita in una sequenza di escape specifica per il terminale usato. In questo modo l'editor poteva funzionare su ognuno dei terminali che era stato inserito nel database *termcap*.

Infine, l'industria si accorse del bisogno di una standardizzazione delle sequenze di escape, quindi fu sviluppato uno standard ANSI; alcuni dei suoi valori sono mostrati in Figura 5.37.

Sequenza di escape	Significato
ESC [n A	Muoversi di <i>n</i> righe verso l'alto
ESC [n B	Muoversi di <i>n</i> righe verso il basso
ESC [n C	Muoversi di <i>n</i> spazi verso destra
ESC [n D	Muoversi di <i>n</i> spazi verso sinistra
ESC [<i>m</i> ; <i>n</i> H	Muovere il cursore in posizione (<i>n</i> , <i>m</i>)
ESC [s J	Pulire lo schermo dalla posizione del cursore [0 alla fine, 1 dall'inizio, 2 tutto]
ESC [s K	Pulire la linea dalla posizione del cursore [0 alla fine, 1 dall'inizio, 2 tutto]
ESC [n L	Inserire <i>n</i> righe dalla posizione del cursore
ESC [n M	Cancellare <i>n</i> righe dalla posizione del cursore
ESC [n P	Cancellare <i>n</i> caratteri dalla posizione del cursore
ESC [n @	Inserire <i>n</i> caratteri dalla posizione del cursore
ESC [n m	Abilitare rendition <i>n</i> [0=normale, 4=grassetto, 5=lampeggiante, 7=rovescio]
ESC M	Scorrere lo schermo all'indietro se il cursore si trova sulla linea più in alto

Figura 5.37 Le sequenze di escape ANSI accettate dai driver di terminale in uscita. ESC denota il carattere ASCII di escape (0x1B) e *n*, *m* e *s* sono parametri numerici opzionali.

Vediamo come queste sequenze di escape possano venire usate da un editor di testi: supponiamo che l'utente digitò un comando che chiede all'editor di cancellare tutta la linea 3 e di eliminare lo spazio tra la linea 2 e la linea 4. L'editor può mandare la seguente sequenza di escape sulla linea seriale al terminale:

ESC [3 ; 1 H ESC [0 K ESC [1 M.

(gli spazi sono usati solo per separare i simboli, ma non vengono trasmessi). Questa sequenza muove il cursore all'inizio della linea 3, cancella la linea, e poi cancella anche la linea che adesso è rimasta vuota, in modo che tutte le linee dalla 4 in poi vengono spostate in su di una linea: quella che era la linea 4 diventa la 3, la 5 diventa la 4, e così via. Sequenze di escape analoghe possono essere usate per aggiungere testo nel mezzo del display, e in modo simile possono essere aggiunte e rimosse parole.

5.7 Interfacce grafiche utente

I PC possono usare interfacce basate sui caratteri ed infatti, per molti anni, MS-DOS, che è basato su caratteri, ha dominato la scena. Comunque oggi giorno la maggior parte dei personal computer usa una **GUI** (*interfaccia grafica utente*, Graphical User Interface).

La GUI è stata inventata da Douglas Engelbart e dal suo gruppo di ricerca dello Stanford Research Institute, ed in seguito è stata copiata dai ricercatori dello Xerox PARC. Un bel giorno, Steve Jobs, cofondatore della Apple, stava visitando il PARC e vide una GUI che girava su un computer Xerox; questo gli diede l'idea di un nuovo computer che diventò l'Apple Lisa. Lisa era troppo costoso e si trasformò in un fallimento commercia-

le, ma il suo successore, il Macintosh, fu un successo enorme. Il Macintosh fu l'ispiratore di Windows della Microsoft, e di altri sistemi basati su GUI.

Una GUI si compone di quattro elementi essenziali, indicati dai caratteri WIMP: queste lettere stanno per Windows (finestre), Icons (icone), Menus (menu) e Pointing device (dispositivo di puntamento). Le finestre sono blocchi rettangolari di area dello schermo, utilizzata per l'esecuzione di programmi; le icone sono piccoli simboli che, se vengono cliccati, producono un qualche tipo di azione; i menu sono liste di azioni tra le quali si può scegliere. Infine, un dispositivo di puntamento è un mouse, un trackball o altri dispositivi hardware che possono essere usati per muovere un cursore in giro per lo schermo o selezionare oggetti.

Il software della GUI può venire implementato sia come codice a livello utente, come nei sistemi UNIX, sia all'interno del sistema operativo stesso, come nel caso di Windows. Nelle sezioni seguenti vedremo l'hardware e il software di input ed output associato alle GUI dei personal computer, con particolare riferimento a Windows, anche se le idee generali sono valide per qualunque sistema basato su GUI.

5.7.1 Tastiera, mouse e hardware di display dei personal computer

I moderni personal computer hanno una tastiera, e un display orientato a bit mappato in memoria; queste componenti sono parte integrante del computer stesso, anche se entrambi, sia la tastiera sia lo schermo, sono completamente separati, con i loro driver specifici.

La tastiera può essere interfacciata tramite porta seriale, porta parallela o una porta USB. Ad ogni azione sui tasti la CPU viene interrotta, e il driver della tastiera preleva il carattere scritto andando a leggere in una porta di I/O; tutto il resto funziona via software, principalmente tramite il driver della tastiera.

Su un Pentium, la tastiera contiene un microprocessore interno, che comunica attraverso una porta seriale specializzata con un controllore che si trova sulla scheda madre: ogni volta che un tasto viene premuto o rilasciato, si genera un interrupt. Inoltre, l'hardware della tastiera fornisce il numero del tasto e non il suo codice ASCII, quindi quando si preme il tasto A, il codice del tasto (30) viene messo in un registro I/O; è compito del driver determinare se è un carattere minuscolo, maiuscolo, se si tratta di CTRL-A, ALT-A, CTRL-ALT-A o qualche altra combinazione. Siccome il driver può dire quali tasti sono stati premuti ma non rilasciati (ad esempio per quanto riguarda SHIFT), ha informazioni sufficienti a svolgere questo compito.

Ad esempio, la sequenza di tasti

PREMI SHIFT, PREMI A, RILASCIA A, RILASCIA SHIFT,

indica una A maiuscola, ma anche

PREMI SHIFT, PREMI A, RILASCIA SHIFT, RILASCIA A,

indica una A maiuscola. Sebbene questa interfaccia di tastiera lasci tutte le parti seccanti nelle mani del software, è estremamente flessibile: per esempio, i programmi utente possono essere interessati a sapere se un numero è stato digitato dai tasti della riga in alto, oppure dalla tastierina numerica, e in linea di principio il driver può fornire questa informazione.

La maggior parte dei PC ha un mouse, o talvolta un trackball, che non è altro che un

mouse a pancia in su. I tipi più comuni di mouse hanno una pallina in gomma che fuoriesce da un foro della base e che ruota se il mouse viene trascinato su una superficie ruvida; nel ruotare, la pallina aziona dei cilindri rotanti di gomma, posti su dei supporti ortogonali. Un movimento in direzione est-ovest fa ruotare i cilindri paralleli all'asse delle y , mentre un movimento in direzione nord-sud fa ruotare i cilindri paralleli all'asse delle x . Ogni volta che il mouse viene spostato di una certa distanza minima in una qualche direzione, oppure ogni volta che viene premuto o rilasciato uno dei suoi tasti, si manda un segnale al computer. La distanza minima è di circa 0,1 mm (anche se può essere modificata via software) e viene talvolta chiamata un **mickey**. I mouse possono avere uno, due o tre tasti, a seconda della stima dei produttori nell'abilità intellettuale dell'utente di gestire più di un tasto.

Il messaggio inviato al computer contiene tre elementi: Δx , Δy , tasti. Il primo elemento rappresenta il cambiamento di posizione nelle x rispetto al messaggio precedente, seguito dal cambiamento di posizione nelle y , ed infine dallo stato dei tasti. Il formato del messaggio dipende dal sistema e dal numero di tasti presenti nel mouse; solitamente occupa 3 byte. La maggior parte dei mouse risponde non più di 40 volte al secondo, quindi il mouse potrebbe aver percorso diversi mickey rispetto al messaggio precedente.

Si noti che il mouse indica unicamente il cambiamento relativo della posizione, e non la posizione assoluta, quindi se viene alzato e riappoggiato con cautela, in modo da non spostare la pallina, non verrà inviato alcun messaggio.

Alcune GUI distinguono tra un singolo click e un doppio click di tasto; se due click avvengono sufficientemente vicini nello spazio (rispetto ai mickey) o nel tempo (in millisecondi), si segnala un doppio click; cosa significhi "sufficientemente vicini" dipende dal software, e comunque entrambi i parametri possono essere impostati dall'utente.

Passiamo adesso dall'hardware di input, all'hardware del display. I dispositivi di display possono essere divisi in due categorie: i dispositivi a **grafica vettoriale** possono accettare ed eseguire comandi come disegnare punti, linee, figure geometriche e testo; i dispositivi a **grafica raster**, invece, rappresentano l'area di uscita come una griglia rettangolare di punti chiamati **pixel**, ognuno dei quali ha un valore di grigio o un colore. Nei primi anni della storia dei calcolatori, i dispositivi a grafica vettoriale erano molto comuni, mentre oggi i plotter sono gli unici dispositivi di questo genere sul mercato; tutto il resto usa **grafica raster**, talvolta chiamata **grafica bitmap**.

I display a grafica raster sono implementati da un dispositivo hardware detto **adattatore grafico**, il quale contiene una memoria speciale detta **RAM video**, che è parte dello spazio di indirizzamento del computer e viene indirizzata dalla CPU come una qualunque altra parte della memoria (si veda Figura 5.38). L'immagine dello schermo viene memorizzata qui, o in modo carattere o in modo bit; in modo carattere ogni byte (o due byte) di RAM video contiene un carattere da visualizzare; in modo bitmap ogni pixel dello schermo viene rappresentato separatamente all'interno della RAM video, con 1 bit per pixel nel caso dei display più semplici in bianco e nero, fino a 24 o più bit per pixel per un display a colori di alta qualità.

Un'altra parte dell'adattatore grafico è un circuito integrato detto **controllore video**; questo circuito prende caratteri o bit dalla RAM video, e genera un segnale video utilizzato per guidare il monitor. Un monitor genera un fascio di elettroni che scandisce lo schermo orizzontalmente, disegnando linee su di esso. Tipicamente, lo schermo ha da 480 a 1024 linee, ognuna formata da 640 fino a 1200 pixel. Il segnale prodotto dal controllore del video modula il fascio di elettroni, decidendo se un dato pixel sarà chiaro o scuro; i monitor a colori hanno 3 fasci, uno per il rosso, uno per il verde e uno per il blu, che vengono modulati in modo indipendente. Anche i display a schermo piatto usano i pixel a tre colori, ma illustrare come lavorino questi schermi va al di là dello scopo di questo libro.

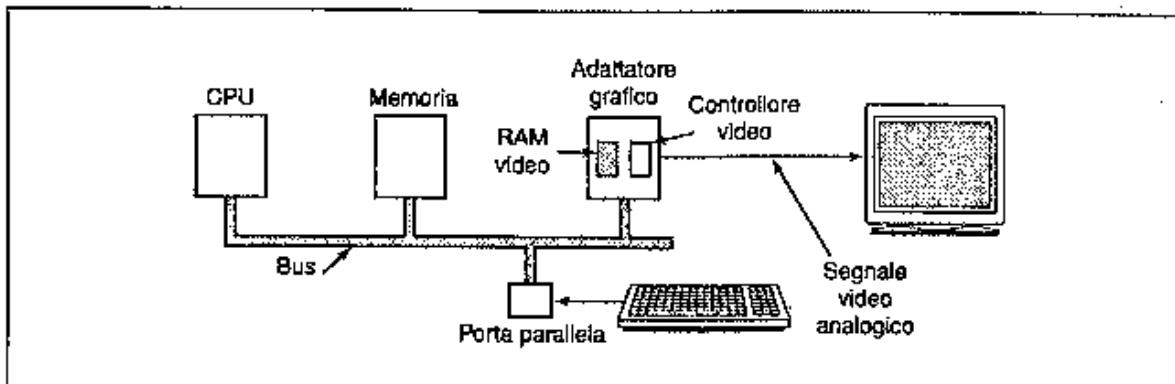


Figura 5.38 Nel caso di display mappati in memoria, il driver scrive direttamente sulla RAM video del display.

I controllori video hanno due modi: modo carattere (usato per il testo semplice) e modo bit (usato per tutto il resto). In modo carattere, il controllore deve adattare ogni carattere in uno spazio di 9 pixel di larghezza per 14 pixel di altezza compreso lo spazio tra i caratteri) e avere 25 linee di 80 caratteri. Il display dovrebbe quindi avere 350 linee di scansione di 720 pixel l'una; ognuna di esse viene ridisegnata da 60 a 100 volte al secondo per renderla più stabile.

Per mostrare il testo sullo schermo, il controllore del video deve ricuperare i primi 80 caratteri nella RAM video, generare 14 linee di scansione, recuperare i successivi 80 caratteri dalla RAM video, generare le 14 linee successive e così via; oppure potrebbe ricuperare ogni carattere una volta per linea di scansione ed eliminare il bisogno di usare il buffer del controllore. Le strutture 9 per 14 pixel per carattere sono tenute in una ROM usata dal controllore video (si può anche usare la RAM per supportare font particolari). La ROM viene indirizzata tramite indirizzi di 12 bit, di cui 8 bit provengono dal codice carattere, e 4 bit specificano la linea di scansione. Gli 8 bit che costituiscono un byte della ROM si riferiscono a 8 pixel; il nono bit tra caratteri viene lasciato vuoto, quindi occorrono $14 \times 80 = 1120$ riferimenti alla RAM video per ogni linea di testo sullo schermo; si fa inoltre lo stesso numero di riferimenti verso la ROM che genera i caratteri.

In Figura 5.39(a) vediamo una porzione della RAM video per un display che opera in modo carattere; ogni carattere sullo schermo, in Figura 5.39(b), occupa due byte nella RAM. Il carattere meno significativo rappresenta il codice ASCII del carattere da visualizzare, mentre il carattere più significativo rappresenta il byte degli attributi, usato per specificare il colore, i caratteri lampeggianti, l'inversione, e così via. Uno schermo di 25 per 80 caratteri ha bisogno di 4000 byte di RAM video, se si usa questo modo.

Il modo bitmap usa lo stesso principio, ma ogni pixel dello schermo viene controllato e rappresentato in modo individuale da 1 o più bit della RAM video; nella configurazione più semplice, quella di un display monocromatico, ogni pixel dello schermo ha un bit corrispondente nella RAM video; all'estremo opposto, ogni pixel dello schermo viene rappresentato da numeri di 24 bit nella RAM video, con 8 bit ciascuno per le intensità di rosso, verde e blu. Questa rappresentazione è detta RGB (Red, Green, Blue) ed è usata perché il rosso, il verde e il blu sono colori primari additivi, a partire dai quali tutti gli altri colori possono essere costruiti, sommando diverse intensità dei colori base.

Le dimensioni degli schermi variano, i più comuni sono 640 x 480 (VGA), 800 x 600 (SVGA), 1024 x 768 (XVGA), 1280 x 1024 e 1600 x 1200; tutti, ad eccezione del 1280 x 1024, hanno un rapporto 4:3, che si adatta al rapporto degli apparecchi televisivi NTSC e produce pixel quadrati. Il 1280 x 1024 avrebbe dovuto essere 1280 x 960, ma sembra che fos-

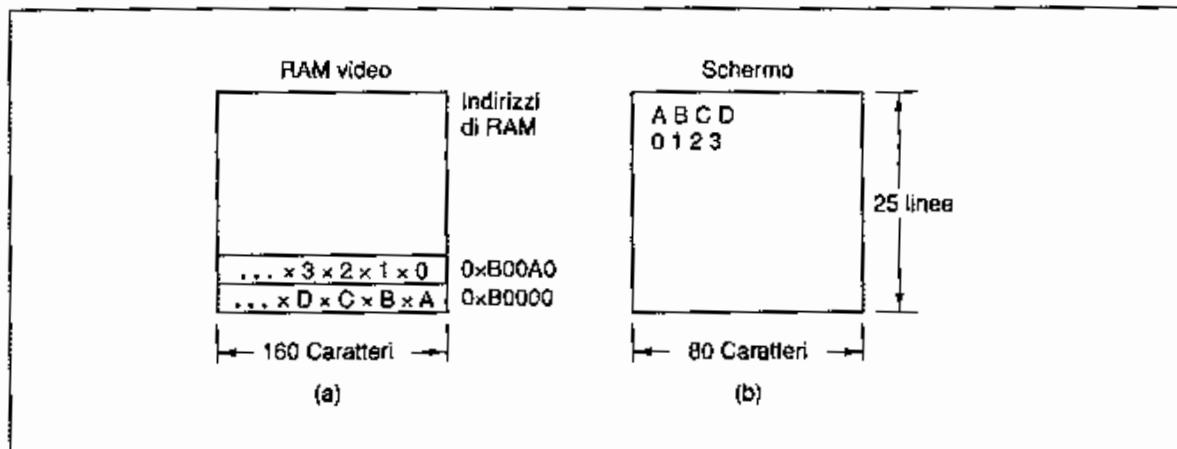


Figura 5.39 (a) Immagine della RAM video per un semplice display monocromatico in modo carattere; (b) Lo schermo corrispondente. Gli x sono byte di attributi.

se impossibile resistere all'attrazione del numero 1024, anche se questa dimensione distorce lievemente i pixel e rende difficile il passaggio da e verso altre scale. A titolo di esempio, un display a colori 768 x 1024 con 24 bit per pixel richiede 2,25 Mb di RAM solo per tenere l'immagine; se si fa un refresh dell'intero schermo 75 volte al secondo, la RAM video deve essere in grado di trasmettere dati in modo continuo a 169 MB/s.

Per evitare di dover gestire immagini così grandi, alcuni sistemi hanno la capacità di fare dei compromessi tra la risoluzione dei colori e la dimensione dell'immagine; nello schema più semplice ogni pixel viene rappresentato da un numero di 8 bit che, invece di indicare il colore, è un indice in una tabella di 256 elementi, ognuno contenente un valore a 24 bit (RGB). Questa tabella, chiamata **paletta (tavolozza) di colori** è spesso mantenuta dall'hardware; essa permette allo schermo di contenere in ogni istante 256 colori qualsiasi; nel momento in cui, ad esempio, cambia il contenuto del settimo elemento della paletta, vengono cambiati tutti i pixel dell'immagine il cui valore era 7. Usare una paletta di colori a 8 bit riduce lo spazio necessario a memorizzare l'immagine sullo schermo da 3 byte per pixel, ad 1 byte per pixel; il prezzo pagato è la diminuzione della risoluzione dei colori. Lo schema di compressione usato dal formato immagini GIF usa una paletta dei colori come quella appena descritta.

È anche possibile usare una paletta di colori a 16 bit per pixel; in questo caso la paletta contiene 65536 ingressi, e quindi si possono usare fino a 65536 colori alla volta; però il guadagno di spazio è minore in quanto si usano 2 byte della RAM video per pixel. Inoltre, se la paletta dei colori è mantenuta in hardware (per evitare di accedere alla memoria per ogni pixel), è richiesto ulteriore hardware dedicato.

È inoltre possibile gestire i colori a 16 bit, memorizzando i valori RGB come numeri di 5 bit, con un avanzo di 1 bit (che può essere assegnato al verde, dato che l'occhio umano sembra essere più sensibile al verde che al rosso o al blu). Questo schema produce risultati simili a quelli del 24 bit per pixel, sacrificando però alcune gradazioni per ogni colore.

5.7.2 Il software di input

Nel momento in cui il driver della tastiera riceve un carattere deve elaborarlo; siccome la tastiera trasmette numeri di tasto invece dei codice carattere usati nel software applicativo, il driver deve effettuare la conversione tra codici, usando una tabella. Non

tutti gli IBM compatibili usano la numerazione dei tasti standard, quindi se il driver vuole supportare queste macchine, deve mappare tastiere diverse con tabelle diverse. Un approccio semplice è quello di compilare una tabella che permetta la conversione dai codici forniti dalla tastiera ai codici ASCII all'interno del driver della tastiera, ma questa scelta non è accettabile per utenti che non siano di lingua inglese: la disposizione dei tasti nelle tastiere cambia da paese a paese, e l'insieme di caratteri ASCII non è adeguato neppure per la maggior parte delle persone dell'emisfero occidentale, dove persone che parlano spagnolo, portoghese e francese hanno bisogno di caratteri accentati e di punteggiatura che non viene usata in inglese. (E lo stesso capita per l'italiano!)

Per rispondere ad esigenze di flessibilità nella disposizione delle tastiere in modo da venire incontro alle diverse lingue, molti sistemi operativi forniscono delle **mappe di caratteri** e delle **pagine di codice**, che possono essere caricate all'avvio del sistema o successivamente e rendono possibile la scelta di una mappa tra codici della tastiera e codici forniti dall'applicazione.

5.7.3 Il software di output in Windows

Il software di output per GUI è un argomento molto ampio. Sono stati scritti un buon numero di libri, ciascuno di più di 1500 pagine, sul solo argomento della GUI di Windows (ad esempio Petzold, 1999; Simon 1997; Rector e Newcomer, 1997). Ovviamente in questa sezione possiamo solo grattare la superficie e presentare pochi concetti fondamentali. Per

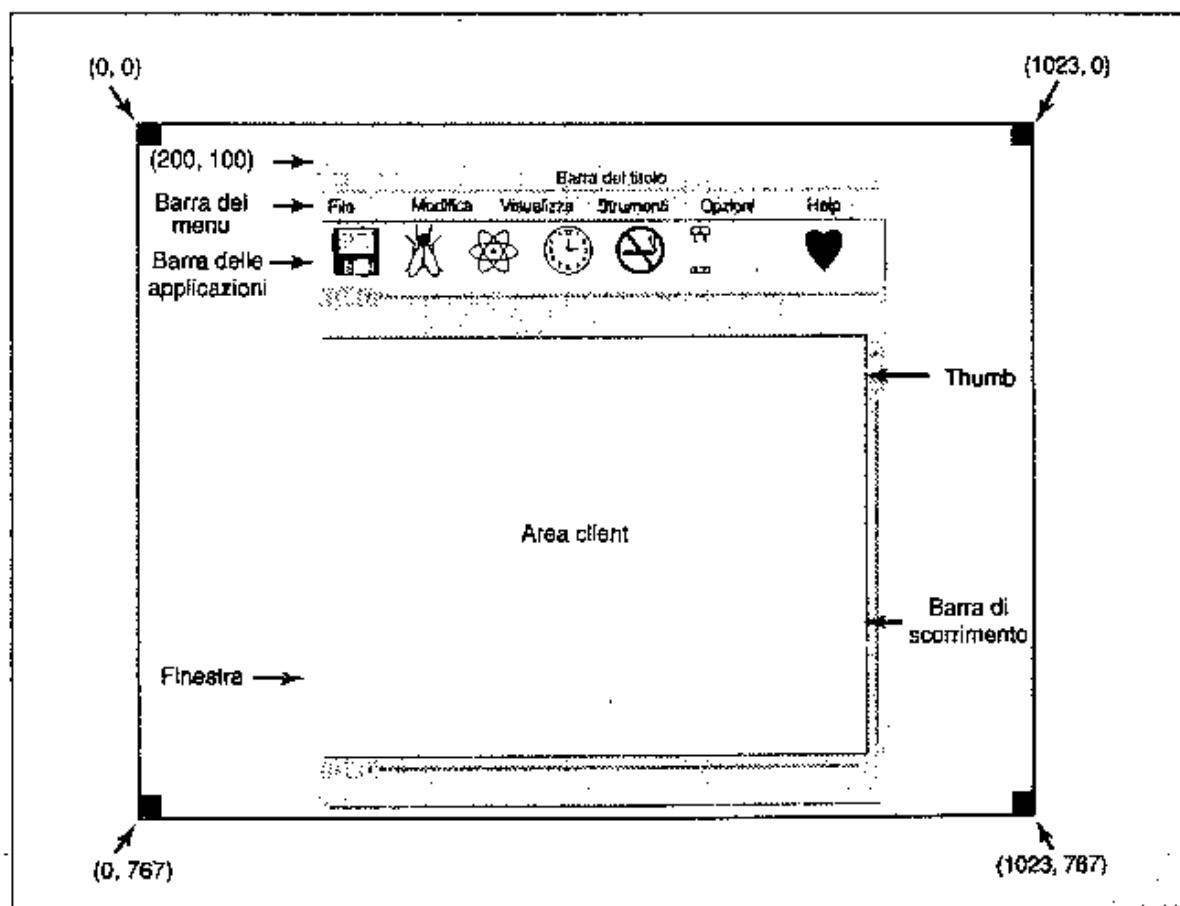


Figura 5.40 Una semplice finestra posta nella posizione [200,100] di un display XGA.

rendere questa descrizione concreta, focalizzeremo l'attenzione sull'API Win32, supportata da tutte le versioni di Windows a 32 bit; il software di output di altre GUI è superficialmente simile, ma i dettagli sono molto diversi.

L'elemento base sullo schermo è un'area rettangolare chiamata **finestra** (*window*); la posizione e la dimensione di una finestra sono rappresentate in modo unico, fornendo le coordinate (in pixel) di una coppia di angoli diagonalmente opposti. Una finestra può contenere una barra di titolo, una barra dei menu, una barra degli strumenti, una o entrambe le barre di scorrimento orizzontale e/o verticale; una finestra tipica è mostrata in Figura 5.40; si noti che il sistema di coordinate di Windows pone l'origine nell'angolo in alto a sinistra, e che l'asse delle *y* è orientato verso il basso, quindi è diverso dal sistema di riferimento del piano Cartesiano, usato in matematica.

Quando viene creata una finestra, i parametri specificano se essa può essere mossa dall'utente, ridimensionata dall'utente, e se si possono usare delle barre di scorrimento: tutto questo ha delle conseguenze enormi sul modo in cui si scrivono i programmi Windows. In particolare, i programmi devono venire informati sugli eventuali cambiamenti nelle dimensioni della loro finestra, e devono essere pronti a ridisegnare il contenuto delle loro finestre in ogni momento, anche quando meno se lo aspettano.

Di conseguenza i programmi Windows sono orientati ai messaggi; le azioni degli utenti riguardanti tastiera o mouse sono catturati da Windows, e convertiti in messaggi al programma proprietario della finestra interessata. Ogni programma ha una coda di messaggi, a cui arrivano tutti i messaggi relativi alle sue finestre; il ciclo principale di un programma consiste nel pescare il messaggio successivo ed elaborarlo tramite una procedura interna relativa a quel tipo di messaggio, ma in alcuni casi Windows stesso può chiamare tali procedure direttamente, ignorando la coda dei messaggi. Questo modello è molto diverso da quello di codice procedurale di UNIX, il quale esegue chiamate di sistema per interagire col sistema operativo.

Per capire meglio questo modello di programmazione, consideriamo l'esempio di Figura 5.41. Qui vediamo lo scheletro di un main di un programma Windows; non è completo e non effettua controllo degli errori, ma contiene abbastanza dettagli da essere utile per il nostro scopo. Comincia con l'inclusione di un file di header, chiamato *windows.h*, che contiene macro, tipi di dati, costanti, prototipi di funzioni e altre informazioni richieste dai programmi Windows.

Il programma inizia con una dichiarazione che definisce il suo nome e i suoi parametri; la macro *WINAPI* è semplicemente un'istruzione al compilatore di usare una certa convenzione di passaggio di parametri, e non ce ne occuperemo. Il primo parametro, *h*, è un'istanza di handle e viene usato per identificare il programma nel resto del sistema; in un certo senso Win32 è orientato ad oggetti, il che significa che il sistema contiene oggetti (ad esempio, programmi, file e finestre) che si trovano in un determinato stato e hanno del codice associato, ossia un insieme di metodi, che operano su quello stato; gli oggetti vengono referenziati utilizzando degli handle, quindi in questo caso *h* identifica il programma. Il secondo parametro è presente solo per compatibilità rispetto a versioni precedenti e non viene più usato. Il terzo parametro, *szCmd*, è una stringa (terminata da uno zero) contenente la linea di comando che ha dato inizio al programma, anche se questo non era stato lanciato da linea di comando; il quarto parametro, *iCmdShow*, dice se la finestra iniziale del programma deve occupare l'intero schermo, parte dello schermo, o nessuna parte dello schermo (solo la barra del task).

Questa dichiarazione illustra una convenzione Microsoft ampiamente usata, chiamata **notazione ungherese**; il nome è una battuta riferita alla notazione polacca, il sistema postfisso inventato dal logico polacco J. Lukasiewicz per rappresentare formule algebriche senza usare precedenze né parentesi. La notazione ungherese è stata inventata da un programmatore ungherese alla Microsoft, Charles Simonyi, e usa i priuni

```

#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE hPrev, char *szCmd, int iCmdShow)
{
    WNDCLASS classfin;           /* Oggetto classe per questa finestra */
    MSG msg;                    /* I messaggi in ingresso sono memorizzati qui */
    HWND hfin;                  /* Handle (puntatore) all'oggetto finestra */

    /* Inizializza classfin */
    classfin.lpfnWndProc = WndProc; /* Dice quale procedura chiamare */
    classfin.lpszClassName = "Nome del programma"; /* Testo per la barra del titolo */
    classfin.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* Carica l'icona del programma */
    classfin.hCursor = LoadCursor(NULL, IDC_ARROW); /* Carica il cursore del mouse */

    RegisterClass(&classfin);      /* Dice a Windows di classfin */
    hfin = CreateWindow(...);     /* Alloca memoria per la finestra */
    ShowWindow(hfin, iCmdShow);   /* Visualizza la finestra sullo schermo */
    UpdateWindow(hwnd);          /* Dice alla finestra di colorarsi */

    while (GetMessage(&msg, NULL, 0, 0)) { /* Prende un messaggio dalla coda */
        TranslateMessage(&msg);       /* Traduce il messaggio */
        DispatchMessage(&msg);       /* Invia il messaggio alla procedura appropriata */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hfin, UINT message, UINT wParam, long lParam)
{
    /* Le dichiarazioni vanno qui. */

    switch (message) {
        case WM_CREATE: ...; return ...; /* Crea una finestra */
        case WM_PAINT: ...; return ...; /* Colora il contenuto della finestra */
        case WM_DESTROY: ...; return ...; /* Distruggi la finestra */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam)); /* default */
}

```

Figura 5.41 Uno scheletro di un main di programma Windows.

caratteri di un identificatore per specificare il tipo; le lettere e i tipi permessi comprendono c (carattere), w (word, parola, ossia un intero di 16 bit senza segno), i (intero di 32 bit con segno), l (long, anch'esso intero di 32 bit con segno), s (stringa), sz (stringa terminata con un byte a zero), p (puntatore), fn (funzione) e h (handle). Quindi szCmd è una stringa terminata da uno zero, mentre iCmdShow è un intero. Molti programmati pensano che codificare in questo modo il tipo nel nome della variabile non ha una vera utilità, e rende solo il codice della Microsoft incredibilmente difficile da leggere; nulla di analogo a questa convenzione è presente in UNIX.

Ogni finestra deve essere associata ad un oggetto classe che definisce le sue proprietà; in Figura 5.41, l'oggetto classe è classfin. Un oggetto di tipo WNDCLASS ha 10 campi, quattro dei quali vengono inizializzati in Figura 5.41; in un programma vero devono essere inizializzati anche gli altri. Il campo più importante è lpfnWndProc, che è un puntatore lungo (32 bit) ad una funzione che gestisce i messaggi diretti alla finestra in questione; gli

altri campi inizializzati qui indicano il nome e l'icona da usare nella barra del titolo, ed il simbolo da usare per il cursore del mouse.

Dopo che la classe *classfin* è stata inizializzata, la *RegisterClass* viene chiamata per passarla a Windows; in particolare, dopo questa chiamata Windows sa quale procedura chiamare quando si verificano determinati eventi che non passano attraverso la coda dei messaggi. La chiamata successiva, *CreateWindow*, alloca la memoria per la struttura dati finestra e ritorna un handle che permette di riferirsi alla finestra stessa in seguito; poi il programma effettua due chiamate ulteriori, per mettere il bordo della finestra sullo schermo, e per riempirla.

A questo punto arriviamo al ciclo principale del programma, che consiste nel prendere un messaggio, farne un certo numero di traduzioni, e passarlo a Windows, il quale invoca *WndProc* che lo elabora. Ci si può chiedere se questo meccanismo non potrebbe essere molto più semplice, e la risposta è sì, ma è stato fatto in questo modo per ragioni storiche e adesso siamo bloccati così.

Seguendo il programma principale troviamo la procedura *WndProc*, che gestisce i vari messaggi che possono essere inviati alla finestra. L'uso di CALLBACK, in modo analogo alla precedente WINAPI, specifica la sequenza da usare per i parametri. Il primo parametro è la handle alla finestra da usare, il secondo parametro è il tipo di messaggio, il terzo e il quarto parametro possono essere usati per fornire informazioni ulteriori, se queste sono necessarie.

I messaggi di tipo WM_CREATE e WM_DESTROY vengono chiamati all'inizio e alla fine del programma, rispettivamente; essi danno al programma la possibilità, ad esempio, di allocare memoria per le strutture dati, e di restituirla alla fine.

Il terzo tipo di messaggio, WM_PAINT, è un'istruzione al programma di riempire la finestra; esso non viene chiamato unicamente nel momento in cui la finestra viene disegnata la prima volta, ma spesso anche durante l'esecuzione del programma. Al contrario dei sistemi basati su testo, in Windows un programma non può ipotizzare che qualunque cosa sia disegnata sullo schermo rimarrà lì finché non sarà esplicitamente rimossa: altre finestre possono venire trascinate al di sopra di questa, possono essere aperti menu o finestre di dialogo che ne copriranno una parte, e così via: quando questi oggetti vengono rimossi, la finestra deve essere ridisegnata. Il modo in cui Windows dice ad un programma di ridisegnare una finestra è attraverso un messaggio WM_PAINT; gli fa anche la gentilezza di mandargli informazioni su quale parte della finestra è stata soprascritta, in caso sia più facile rigenerare solo quella parte invece dell'intera finestra.

Ci sono due modi in cui Windows può chiedere ad un programma di fare qualcosa: un modo è quello di inviare un messaggio nella sua coda; questo metodo è usato per l'input da tastiera e da mouse, e per timer che sono scaduti. L'altro modo, attraverso un messaggio ad una finestra, costringe Windows a chiamare *WndProc* direttamente. Questo metodo viene usato per tutti gli altri eventi; siccome Windows sa quando un messaggio è stato completamente elaborato, può evitare di emettere una nuova chiamata finché la precedente non è completata, ed in questo modo si evitano condizioni di corsa critica.

Ci sono molti altri tipi di messaggi: per evitare comportamenti erratici nel caso arrivi un segnale inatteso, è meglio chiamare *DefWindowProc* alla fine di *WndProc*, e lasciare che il gestore di default gestisca tutti gli altri casi.

Riassumendo, un programma Windows normalmente crea una o più finestre con un oggetto classe per ognuna di esse; ad ogni programma è associata una coda di messaggi e una serie di procedure di gestione. In pratica, il comportamento del programma è guidato dagli eventi, che vengono elaborati dalle procedure di gestione. Questo modello è molto diverso dalla visione procedurale scelta da UNIX.

Il vero e proprio disegno dello schermo è gestito da un pacchetto che si compone di

centinaia di procedure che sono messe insieme a formare la **GDI** (*dispositivo di interfaccia grafica*, Graphics Device Interface); essa può gestire testo e ogni tipo di grafica, ed è progettata per essere indipendente dalla piattaforma e dal dispositivo. Prima che un programma possa disegnare in una finestra, ha bisogno di acquisire un **contesto di dispositivo**, ossia una struttura dati interna che contiene le proprietà della finestra, quali il font corrente, il colore del testo, il colore dello sfondo, eccetera. La maggior parte delle chiamate alla GDI usano questo contesto di dispositivo, o per disegnare o per prendere o modificare le proprietà.

Esistono vari modi di acquisire il contesto di dispositivo; un semplice esempio di acquisizione e uso è il seguente:

```
hdc = GetDC(hfin);
TextOut(hdc, x, y, pTesto, iLung);
ReleaseDC(hfin, hdc).
```

Il primo comando riceve l'handle del contesto di dispositivo, *hdc*; il secondo usa il contesto di dispositivo per scrivere una linea di testo sullo schermo, specificando le coordinate (*x*, *y*) da cui la stringa deve iniziare, un puntatore alla stringa stessa e la sua lunghezza. La terza chiamata rilascia il contesto di dispositivo, per indicare che il programma per il momento ha completato la fase di disegno. Si noti che *hdc* è usata in modo analogo ad un descrittore di file UNIX ed inoltre che *ReleaseDC* contiene informazioni ridondanti (l'uso di *hdc* specifica una finestra in modo unico). L'uso di informazioni ridondanti che non abbiano alcun valore effettivo è comune in Windows.

Un altro dettaglio interessante è che quando *hdc* viene acquisito in questo modo, il programma può solo scrivere nell'area client della finestra, non nella barra del titolo o in altre parti della finestra. Internamente, nella struttura dati del contesto di dispositivo, viene mantenuta una regione di raccolta, e ogni aggiunta al di fuori di questa regione viene ignorata. Comunque esiste un altro modo di acquisire il contesto di dispositivo, attraverso *GetWindowDC*, che imposta la regione di lavoro all'intera finestra; altre chiamate restringono la regione di lavoro in altri modi. Avere molte chiamate che fanno quasi la stessa cosa è un'altra caratteristica di Windows.

Una trattazione completa della GDI non può essere proposta nel contesto di questo libro; il lettore interessato è rimandato alle citazioni fornite in precedenza, che possono fornire informazioni aggiuntive. Ciononostante, qualche parola sulla GDI è doverosa, vista la sua importanza; la GDI ha varie procedure per ottenere e rilasciare contesti di dispositivo, ottenere informazioni dai contesti di dispositivo, ottenere ed impostare attributi dei contesti di dispositivo (ad esempio il colore dello sfondo), manipolare oggetti GDI quali penni, pennelli e font, ognuno dei quali ha i propri attributi. Infine, naturalmente, ci sono un grande numero di chiamate che effettivamente disegnano sullo schermo.

Le procedure di disegno ricadono in quattro categorie: disegnare linee e curve, disegnare aree piene, gestire bitmap e visualizzare testo. Abbiamo già visto un esempio di visualizzazione di testo, quindi adesso daremo un'occhiata alle altre categorie. La chiamata

```
Rectangle(hdc, xsin, yalto, xdes, ybasso);
```

disegna un rettangolo pieno i cui angoli sono (*xsin*, *yalto*) e (*xdes*, *ybasso*). Ad esempio

```
Rectangle(hdc, 2, 1, 6, 4);
```

disegnerà il rettangolo mostrato in Figura 5.42. Lo spessore e il colore della linea, oltre che

il colore del riempimento, sono presi dal contesto di dispositivo. Le altre chiamate GDI hanno un aspetto simile.

Bitmap

Le procedure GDI sono esempi di grafica vettoriale, e vengono usate per inserire figure geometriche e testo sullo schermo. Possono essere scalati facilmente per schermi più grandi o più piccoli (supponendo che il numero di pixel sullo schermo sia lo stesso) e sono relativamente indipendenti dal dispositivo. Una collezione di chiamate a procedure GDI può essere assemblata in un file che descrive un disegno complesso; questo file è chiamato **metafile** di Windows ed è ampiamente usato per trasmettere disegni da un programma Windows ad un altro, in file di estensione **.wmf**.

Molti programmi Windows lasciano che l'utente copi (parte di) un disegno e lo metta nella clipboard (bacheca) di Windows; l'utente può quindi aprire un altro programma e copiare il contenuto della clipboard in un altro documento. Un modo di ottenere questo risultato è il seguente: il primo programma rappresenta il disegno come un metafile Windows e lo mette nella clipboard in formato **.wmf**, ma esistono altri modi di procedere.

Non tutte le immagini che i computer manipolano possono essere generate usando grafica vettoriale: fotografie e video, per esempio, non usano grafica vettoriale, ma vengono scannerizzati sovrapponendo loro una griglia. I valori medi di rosso, verde, blu di ogni quadrato sottostante la griglia vengono salvati come valori del pixel corrispondente; il file prodotto è chiamato **bitmap**. Ci sono molti modi in Windows per manipolare le bitmap.

Un altro uso delle bitmap riguarda il testo: un modo per rappresentare un carattere specifico in qualche font è tramite una piccola bitmap. Aggiungere testo sullo schermo diventa quindi un problema di spostamento di bitmap.

Un modo generale per usare le bitmap è attraverso la procedura **bitblt**, che viene chiamata nel seguente modo:

```
bitblt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

Nella sua forma più semplice, copia una bitmap da un rettangolo in una finestra in un rettangolo di un'altra finestra (o della stessa); i primi tre parametri specificano la finestra

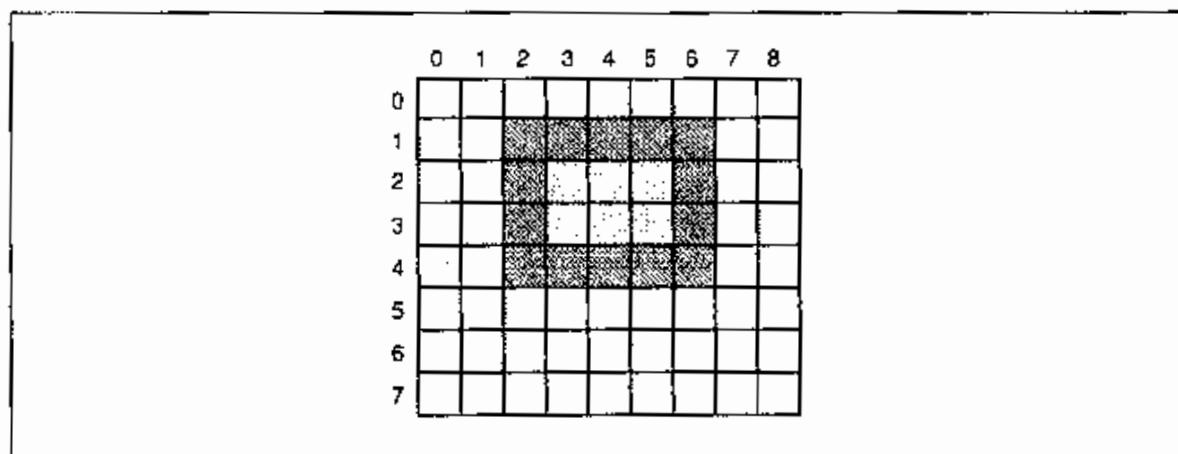


Figura 5.42 Esempio di rettangolo disegnato usando la chiamata **Rectangle**. Ogni quadratino rappresenta un pixel.

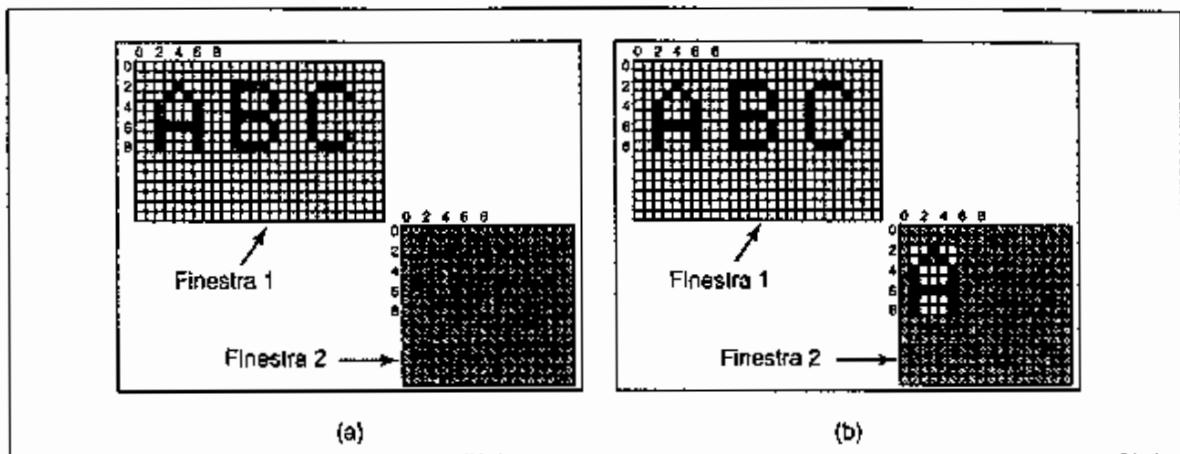


Figura 5.43 Copia di bitmap tramite la procedura bitblt. (a) prima, (b) dopo.

destinazione e la sua posizione, i successivi due larghezza e altezza, segue la finestra sorgente e la sua posizione. Si noti che ogni finestra ha il proprio sistema di coordinate, dove (0,0) è l'angolo in alto a sinistra della finestra. L'ultimo parametro verrà descritto nel seguito. L'effetto di

```
Bitblt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

è mostrato in Figura 5.43: si noti che è stata copiata l'intera area 5x7 occupata dalla lettera A, incluso lo sfondo.

Bitblt può fare molto di più che copiare bitmap: l'ultimo parametro dà la possibilità di eseguire operazioni booleane per combinare la bitmap sorgente e quella destinazione; ad esempio, la bitmap sorgente può essere messa in OR con la destinazione in modo da fonderle insieme, oppure può essere messa in OR ESCLUSIVO, in modo da mantenere le caratteristiche sia della sorgente che della destinazione.

Un problema con le bitmap è che non possono essere scalate: un carattere 8 x 12 su un display 640 x 480 sembrerà ragionevole, ma se questa bitmap è copiata su una pagina stampata a 1200 punti/pollice, che equivale a 10200 bit x 13200 bit, la larghezza del carattere (8 pixel) diventerà di 8/1200 pollici ossia 0,17 mm. Inoltre la copia tra dispositivi con diverse proprietà di colore, o tra dispositivi monocromatici e dispositivi a colori non funziona bene.

Per questa ragione Windows supporta una struttura dati chiamata DIB (*bitmap indipendente dai dispositivi*, Device Independent Bitmap). I file che usano questo formato hanno estensione .bmp, hanno le intestazioni di file, le informazioni, e una tabella di colori che precede i pixel. Queste informazioni rendono più semplice il passaggio delle bitmap tra diversi dispositivi.

Font

Nelle versioni di Windows precedenti alla 3.1 i caratteri erano rappresentati come bitmap e copiati sullo schermo o sulla stampante tramite la *bitblt*; il problema con questo approccio è, come abbiamo appena visto, che una bitmap di dimensione ragionevole per lo schermo può essere troppo piccola per la stampante; inoltre occorre una diversa bitmap per ogni carattere e in ogni dimensione ossia, in altre parole, data la bitmap per la A

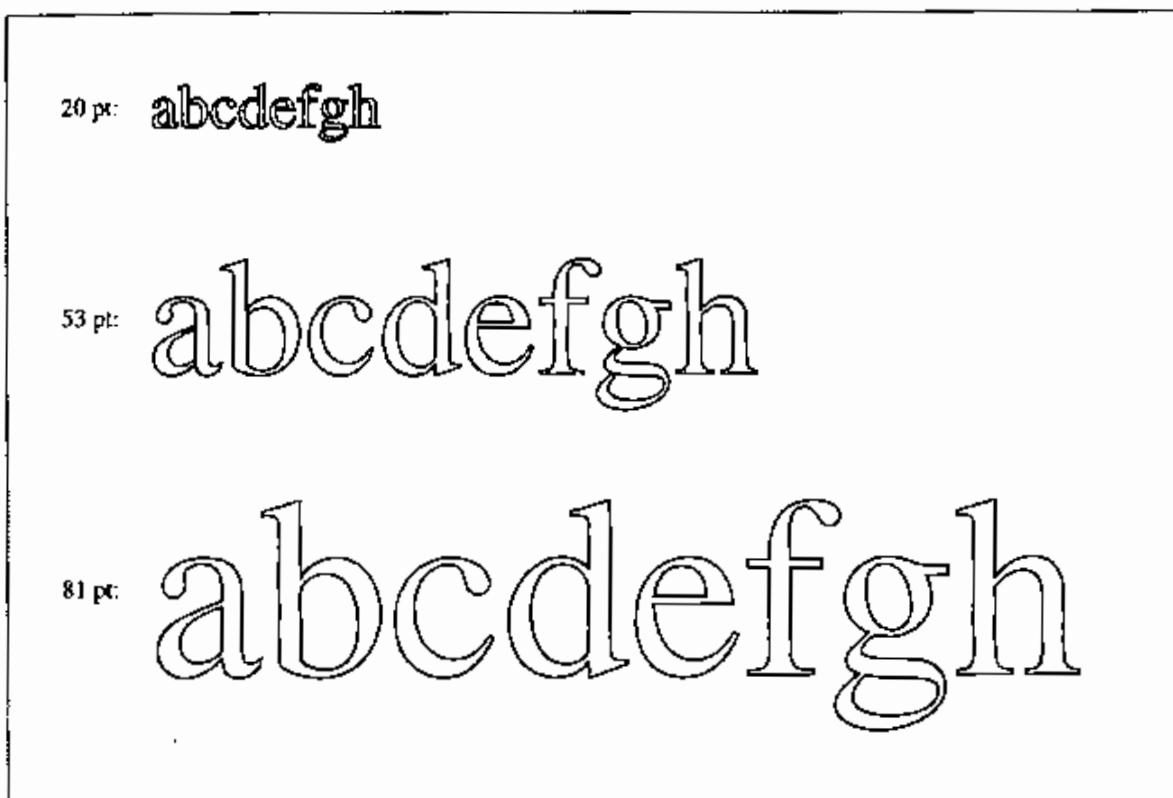


Figura 5.44 Alcuni esempi di contorni di caratteri a scale diverse.

in una dimensione di 10 punti, non c'è modo di calcolare la bitmap di A in dimensione 12 punti. Siccome ogni carattere di ogni font potrebbe servire, in una gamma di dimensioni che va da 4 a 120 punti, servirebbe un grande numero di bitmap; si trattava quindi di un sistema un po' troppo pesante per gestire il solo testo.

La soluzione è arrivata con l'introduzione dei font TrueType, che non sono bitmap ma schemi di caratteri. Ogni carattere TrueType è definito da una sequenza di punti intorno al suo perimetro, tutti relativi all'origine (0,0). Usando questo sistema, è facile ingrandire o rimpicciolire i caratteri, basta moltiplicare ogni coordinata dello stesso fattore di scala e, in questo modo, un carattere TrueType può essere scalato ad ogni dimensione, anche a quelle frazionarie. Una volta trasformato il carattere nella giusta dimensione, i punti vengono uniti usando il ben noto algoritmo "unire i puntini" che viene insegnato all'asilo (si noti che negli asili moderni si usano le spline per ottenere risultati più lisci); dopo aver completato il contorno, il carattere può essere riempito. Un esempio di caratteri scalati a tre diverse dimensioni di punti è mostrato in Figura 5.44.

Una volta che si sono riempiti i caratteri nella loro forma matematica, essi possono essere rasterizzati, ossia convertiti in bitmap della risoluzione voluta. Poiché la rasterizzazione segue la scalatura, siamo sicuri che i caratteri mostrati sullo schermo e quelli che appaiono sul foglio stampato, saranno il più simili possibile, differendo solo per un errore di quantizzazione. Per migliorare ulteriormente la qualità, è possibile inserire dei suggerimenti relativi ad ogni carattere, sul come fare la rasterizzazione. Per esempio entrambe le gambette in alto della T devono essere identiche, cosa che potrebbe non risultare vera a causa di errori di arrotondamento.

5.8 Terminali di rete

I terminali di rete sono usati per collegare utenti remoti a computer attraverso la rete, o su una rete locale (LAN, Local Area Network) o su una rete globale (WAN, Wide Area Network). Esistono due diverse filosofie secondo le quali un terminale di rete dovrebbe funzionare; secondo la prima il terminale dovrebbe avere una grande potenza di calcolo e di memoria allo scopo di eseguire protocolli complessi per la compressione dei dati da spedire attraverso la rete (un protocollo è un insieme di richieste e risposte su cui un mittente ed un destinatario si trovano d'accordo, allo scopo di comunicare sulla rete o su altre interfacce). Secondo la seconda teoria, il terminale dovrebbe essere estremamente semplice, essenzialmente dovrebbe visualizzare i pixel e non pensare troppo, allo scopo di renderlo molto economico. Nelle due sezioni seguenti discuteremo un esempio per ognuna di queste filosofie, prima esamineremo il sofisticato sistema X Window e poi analizzeremo il sistema di terminale minimale SLIM.

5.8.1 Il sistema X Window

Il meglio tra i terminali intelligenti è un terminale che contiene una CPU potente come quella del computer centrale, insieme con megabyte di memoria, una tastiera e un mouse. Un terminale di questo tipo è il **terminale X**, che lavora con il sistema X Window (spesso chiamato semplicemente X), sviluppato dal M.I.T. come parte del progetto Athena. Un terminale X è un computer dotato di software X, che interagisce con i programmi eseguiti sui computer remoti.

Il programma all'interno del terminale X che colleziona input dalla tastiera o dal mouse, e accetta comandi da computer remoti, è chiamato **server X**. Esso deve tenere traccia di quale finestra sia attiva in un determinato momento (dove si trova il puntatore del mouse), in modo da sapere a quale client inviare l'input proveniente dalla tastiera; inoltre comunica attraverso la rete con **client X** eseguiti su host remoti. Esso invia loro gli input provenienti da mouse e tastiera, e accetta da essi comandi di display.

Può sembrare strano avere il server X all'interno di un terminale, e i client su un host remoto, ma il lavoro del server X è quello di visualizzare bit, quindi ha senso che si trovi vicino all'utente. Dal punto di vista del programma, è il client a dire al server cosa fare, ad esempio visualizzare testo o figure geometriche; il server (sul terminale) esegue quello che gli viene chiesto, come fanno tutti i server. L'organizzazione di client e server è mostrato in Figura 5.45.

È anche possibile eseguire il sistema X Window sopra UNIX o altri sistemi operativi, ed infatti molti sistemi UNIX eseguono X come il loro sistema a finestre standard, anche su macchine fuori rete o per accedere a computer remoti attraverso Internet. Quello che il sistema X Window definisce realmente è il protocollo tra il client X e il server X, come mostrato in Figura 5.45; non fa differenza che il client e il server risiedano sulla stessa macchina, siano a 100 metri di distanza l'uno dall'altro su una rete locale, o a migliaia di chilometri di distanza e corressi tramite Internet. Il protocollo e il funzionamento del sistema sono uguali in tutti questi casi.

X è semplicemente un sistema di finestre, non è una GUI completa. Per ottenere una GUI completa occorrono altri strati di software al di sopra di X; uno strato è **Xlib**, un insieme di procedure di libreria per accedere alle funzionalità di X. Queste procedure formano la base del sistema X Window e sono quello che esamineremo fra poco, ma sono troppo primitive per essere utilizzate direttamente dalla maggior parte dei programmati. Per esempio, ogni click del mouse viene notificato separatamente, quindi accorgersi

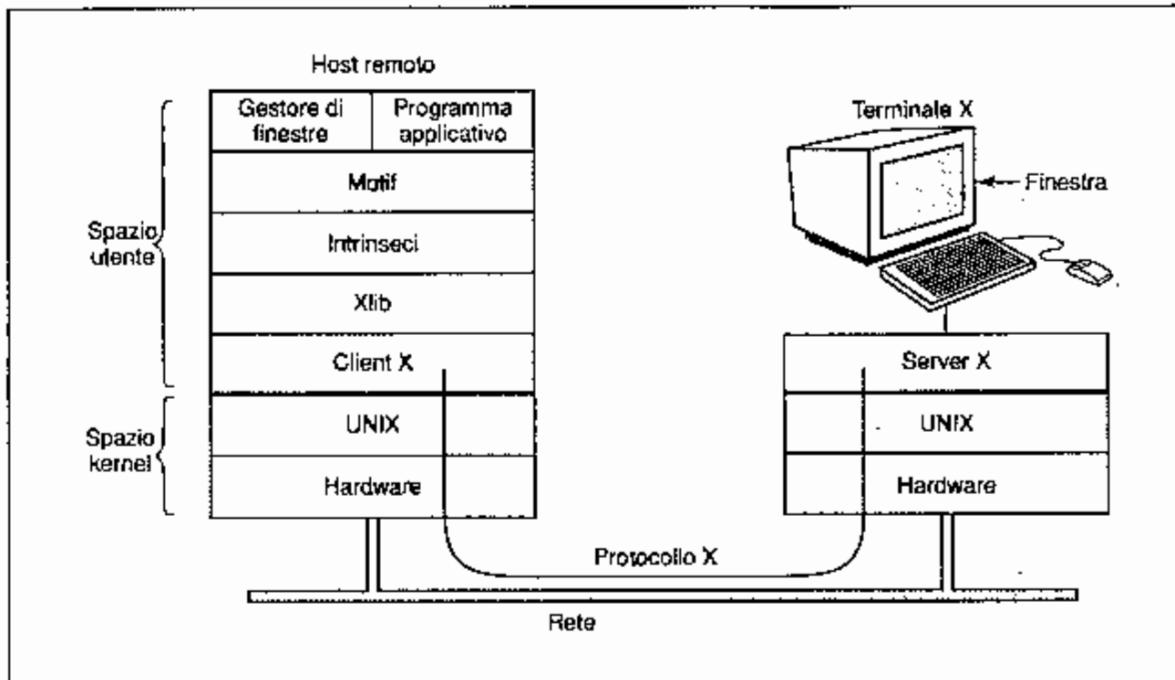


Figura 5.45 Client e server nel sistema X Window del M.I.T.

che due click sono in realtà un doppio click è un'operazione che deve essere gestita al di sopra di Xlib.

Per rendere più facile la programmazione con X, viene fornito, come parte di X, un toolkit che si compone degli **Intrinsic**; questo strato gestisce i tasti, le barre di scorrimento, e altri elementi di una GUI chiamati **widget**; ma per ottenere una GUI vera e propria, con un aspetto uniforme, occorre uno strato ulteriore; il più diffuso è chiamato **Motif**. La maggior parte delle applicazioni fanno chiamate direttamente a Motif e non a Xlib.

E anche il caso di notare che la gestione delle finestre non è parte di X; la decisione di lasciarla al di fuori è stata completamente voluta e lasciata a un processo X client separato, chiamato **gestore di finestre** (*window manager*), che controlla la creazione, la cancellazione e lo spostamento di finestre sullo schermo. Per gestire le finestre, esso invia comandi al server X; spesso risiede sulla stessa macchina del client X, ma in teoria potrebbe venire eseguito da qualunque altra parte.

Questa progettazione modulare, che si compone di diversi strati e molti programmi, rende X altamente portabile e flessibile; esso è stato fatto migrare sulla maggior parte delle versioni di UNIX, incluse Solaris, BSD, AIX, Linux e così via, fornendo agli sviluppatori di applicazioni un'interfaccia standard per molte piattaforme; è stato fatto migrare anche su altri sistemi operativi. Viceversa, in Windows, il sistema a finestre e la GUI sono stati mischiati nella GDI e sono stati messi nel **kernel**, rendendoli molto più difficili da gestire. Ad esempio la GUI di Windows 98 è ancora fondamentalmente a 16 bit, dopo che da più di un decennio i processori della Intel sono a 32 bit.

A questo punto, diamo un'occhiata ad X a livello di Xlib. Quando un programma X viene avviato, esso apre una connessione con uno o più server X – che possiamo chiamare **workstation** (stazioni di lavoro) anche se possono essere collocati sulla stessa macchina del programma X stesso. X considera questa connessione come affidabile, nel senso che messaggi persi o duplicati sono gestiti dal software di rete, ed esso non deve quindi preoccuparsi di errori di comunicazione. Solitamente tra client e server si usa TCP/IP.

Ci sono quattro tipi di messaggi che viaggiano lungo la connessione:

1. Comandi di disegno dal programma alla workstation.
2. Risposte della workstation alle richieste del programma.
3. Annunci di eventi prodotti da tastiera, mouse e altro.
4. Messaggi di errore.

La maggior parte dei comandi di disegno sono inviati dal programma alla workstation come messaggi unidirezionali, e non ci si aspetta alcuna risposta. La ragione di questo tipo di progettazione risiede nel fatto che, quando i processi client e server si trovano su macchine diverse, può passare un certo periodo di tempo prima che i comandi raggiungano il server e possano essere eseguiti: bloccare l'applicazione per questo tempo la rallenterebbe più del necessario. Se però il programma ha bisogno di informazioni da parte della workstation, semplicemente deve aspettare che queste informazioni diventino disponibili.

Come Windows, anche X è profondamente guidato dagli eventi: gli eventi fluiscano dalla workstation al programma, solitamente in risposta a qualche azione umana come premere i tasti, muovere il mouse, o scoprire una finestra. Ogni messaggio di evento è di 32 byte: il primo byte descrive il tipo di evento e i successivi 31 forniscono informazioni aggiuntive. Esistono svariate decine di eventi, ma un programma riceve solo messaggi riguardanti eventi che esso è disposto a gestire; per esempio, se un programma non è interessato al rilascio dei tasti, non riceverà messaggi che riguardano tale tipo di eventi. Come in Windows, gli eventi sono messi in una coda, e i programmi leggono gli eventi da tale coda, ma, a differenza di Windows, il sistema operativo non chiama mai personalmente procedure che appartengono all'applicazione, e anzi, non sa nemmeno quali procedure gestiscono un particolare evento.

Il concetto principale di X è la **risorsa**. Una risorsa è una struttura dati che contiene certe informazioni; i programmi applicativi creano le risorse sulle workstation. Le risorse possono essere condivise tra più processi della workstation; esse hanno solitamente una vita breve e non sopravvivono al riavvio della workstation. Risorse tipiche sono le finestre, i font, le mappe di colori (palette), e di pixel (bitmap), i cursori, i contesti grafici. Questi ultimi vengono usati per associare proprietà a finestre, e come idea sono simili ai contesti di dispositivo di Windows.

Uno scheletro base e incompleto di un programma X è mostrato in Figura 5.46. Esso inizia includendo alcuni file necessari e dichiarando alcune variabili, poi si connette al server X specificato come parametro in *XOpenDisplay*. A questo punto alloca una risorsa finestra, e memorizza l'handle relativo a questa finestra in *fin*, in pratica fa alcune inizializzazioni. Dopo di ciò informa il gestore delle finestre dell'esistenza di una nuova finestra da gestire.

La chiamata a *XCreateGC* crea un contesto grafico nel quale vengono immagazzinate le proprietà della finestra; in un programma più complesso dovrebbero essere inizializzate qui. Nel comando successivo, la chiamata a *XSelectInput* dice al server X quali eventi il programma è preparato a gestire: in questo caso il programma è interessato ai click del mouse, alla pressione dei tasti della tastiera, ed alle finestre che vengono scoperte; in pratica un vero programma dovrebbe essere interessato anche ad altri eventi. Infine la chiamata a *XMapRaised* mappa la nuova finestra sullo schermo, come finestra più in alto: a questo punto la finestra diventa visibile sullo schermo.

Il ciclo principale del programma si compone di due comandi e, dal punto di vista logico, è molto più semplice dell'equivalente ciclo in Windows. Il primo comando prende un evento e il secondo lo classifica secondo il tipo di evento per dare inizio all'elaborazione. Quando qualche evento indica che il programma ha finito, *attivo* è impostato a 0

```

#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;           /* * Identificatore del server */
    Window fin;            /* * Identificatore della finestra */
    GC gc;                /* * Identificatore del contesto grafico */
    XEvent evento;         /* * Memoria per un evento */
    int attivo = 1;

    disp = XOpenDisplay("Nome_display"); /* Collegamento al server */
    fin = XCreateSimpleWindow(disp, ...); /* Alloca memoria per la nuova finestra */
    XSetStandardProperties(disp, ...);   /* Annuncia la finestra al gestore delle finestre */
    gc = XCreateGC(disp, fin, 0, 0);     /* Crea un contesto grafico */
    XSelectInput(disp, fin, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, fin);             /* Visualizza la finestra; invia l'evento Expose */

    while (attivo) {
        XNextEvent(disp, &evento);      /* Prendi il prossimo evento */
        switch (evento.type) {
            case Expose: ...; break;   /* Ricolora la finestra */
            case ButtonPress: ...; break; /* Elabora il click del mouse */
            case Keypress: ...; break;  /* Elabora l'input da tastiera */
        }
    }

    XFreeGC(disp, gc);           /* Rilascia il contesto grafico */
    XDestroyWindow(disp, fin);   /* Dealloca lo spazio di memoria della finestra */
    XCloseDisplay(disp);         /* Interrompi la connessione di rete */
}

```

Figura 5.46 Scheletro di un programma applicativo X Windows.

e il ciclo termina; prima di uscire il programma rilascia il contesto grafico, la finestra e la connessione.

È il caso di accennare al fatto che le GUI non piacciono a tutti; molti programmati preferiscono le classiche interfacce orientate a linea di comando, del tipo discusso nelle Sezione 5.6.2. X si prende cura di ciò tramite un programma client chiamato *xterm*, che emula il vecchio terminale intelligente VT102 ed è completo di tutte le sequenze di escape. Quindi programmi editor quali *vi* e *emacs* e altro software che usa termcap, possono lavorare in queste finestre senza bisogno di modifiche.

5.8.2 Il terminale di rete SLIM

Nel corso degli anni, i paradigmi di elaborazione principali hanno oscillato tra l'elaborazione centralizzata e distribuita: i primi computer, ad esempio ENIAC, erano infatti personal computer anche se grandi, in quanto solo una persona per volta poteva utilizzarli. In seguito arrivarono i sistemi a condivisione del tempo (timesharing), nei quali

molti utenti remoti connessi da terminali semplici, potevano condividere un unico grande computer centrale. Successivamente arrivò l'era dei PC, nella quale gli utenti possedevano nuovamente il loro computer personale.

Anche se il modello di PC decentralizzati ha i suoi vantaggi, ha anche alcuni svantaggi che si sta appena iniziando a considerare seriamente. Probabilmente il più grande problema è che ogni PC ha un disco fisso grande, e del software complesso che deve essere mantenuto; per esempio, quando esce una nuova versione di un sistema operativo, occorre una grande quantità di lavoro per effettuare un aggiornamento separato per ogni macchina. In molte grandi aziende, i costi di questo genere di manutenzione del software fanno sparire i costi di hardware e software. Per utenti privati, la manutenzione è tecnicamente gratuita, ma poche persone sono in grado di portarla avanti in modo corretto e pochissime persone la trovano ancora divertente. Con un sistema centralizzato, solo una o poche macchine devono essere aggiornate, e queste macchine hanno uno staff di esperti che svolgono il lavoro.

Un problema collegato è che gli utenti dovrebbero fare backup regolari dei loro file system grandi gigabyte, ma pochi di essi lo fanno, ed in caso di disastro, ne consegue un gran lamentarsi e disperarsi. Con un sistema centralizzato, i backup possono essere fatti ogni notte in modo automatico.

Un altro vantaggio risiede nel fatto che la condivisione delle risorse è più facile nel caso di sistemi centralizzati. Un sistema con 64 utenti remoti, ognuno con 64 MB di RAM avrà la maggior parte di questa RAM inattiva per la maggior parte del tempo. In un sistema centralizzato con 4 GB di RAM, non capita mai che alcuni utenti richiedano contemporaneamente molta RAM, ma non possono ottenerla perché la RAM di cui hanno bisogno risiede sul PC di qualcun altro. Lo stesso ragionamento è valido per lo spazio disco e altre risorse.

Probabilmente, una conclusione equa sta nel dire che la maggior parte degli utenti vogliono computer interattivi ad alte prestazioni, ma non hanno veramente voglia di amministrare il loro computer; questo ha portato i ricercatori a riesaminare la condivisione di tempo tramite terminali "stupidi" (oggi educatamente chiamati *client sottili*), che soddisfino le aspettative attuali. X era un passo in questa direzione, ma un server X è comunque un sistema complesso costituito da megabyte di software che deve essere aggiornato ogni tanto. Il sacro Graal sarebbe un sistema di elaborazione interattivo ad

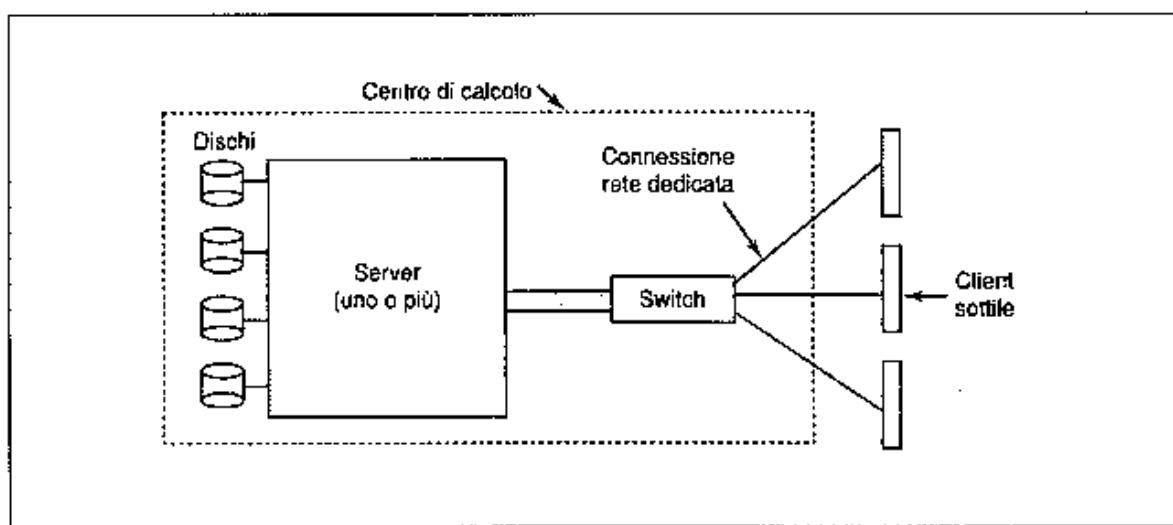


Figura 5.47 L'architettura di un sistema di terminali SLIM.

alte prestazioni, in cui la macchina dell'utente non abbia alcun software; la cosa interessante è che questo obiettivo è raggiungibile. Nel seguito descriviamo uno di questi sistemi, sviluppato dai ricercatori della Sun Microsystems e della Stanford University, e oggi commercializzato dalla Sun (Schmidt et al. 1999).

Il sistema è chiamato **SLIM** (in inglese *slim* significa magro) che sta per **S**tateless **L**ow-level **I**nterface **M**achine (macchina senza stati con interfaccia a basso livello). L'idea è basata sulla tradizionale condivisione del tempo centralizzata, come mostrato in Figura 5.47. Le macchine client sono semplicemente degli stupidi display a bitmap 1280x1024 con una tastiera, un mouse e nessun software personalizzato, e seguono molto lo spirito dei vecchi terminali intelligenti orientati a carattere, che avevano dei meccanismi per interpretare i codici di escape, ma nessun altro software. Come i vecchi terminali orientati a carattere, l'unico controllo che l'utente possa applicare è il tasto di accensione o spegnimento. I terminali di questo tipo, con poca capacità di elaborazione, sono detti client sottili.

Il modello più semplice – nel quale il server invia bitmap attraverso la rete ai client SLIM 60 volte al secondo – non può funzionare: ha bisogno di circa 2 GB al secondo di banda di rete, che è troppo per le attuali reti. Il secondo modello più semplice – che memorizza l'immagine dello schermo in un buffer del terminale, e lo aggiorna 60 volte al secondo – è molto più promettente. In particolare, se il server centrale mantiene una copia di ogni buffer di terminale ed invia solo gli aggiornamenti (ossia i cambiamenti avvenuti) quando necessario, i requisiti di banda diventano piuttosto modesti. Ecco come lavorano i client sottili SLIM.

A differenza del protocollo X, che ha centinaia di messaggi complessi per gestire finestre, disegnare figure geometriche, visualizzare testo in molti font diversi, il protocollo SLIM ha solo cinque messaggi di display, elencati in Figura 5.48 (ci sono inoltre un piccolo numero di messaggi di controllo che non sono stati inclusi nell'elenco). Il primo, SET, sostituisce un rettangolo del buffer con pixel nuovi; ogni pixel sostituito richiede 3 byte nel messaggio, che servono a specificare il suo valore di colore completo (a 24 bit). In teoria questo messaggio è sufficiente; gli altri messaggi esistono solo in qualità di ottimizzazione.

Il messaggio FILL riempie tutto un rettangolo con un singolo valore; viene usato per riempire gli sfondi uniformi. Il messaggio BITMAP riempie tutto un rettangolo ripetendo un certo pattern contenuto in una bitmap fornita dal messaggio stesso. Questo comando è utile per riempire sfondi non uniformi, ma con una certa texture.

Il messaggio COPY chiede al terminale di copiare un rettangolo all'interno del buffer in un altro punto del buffer; è molto utile per scorrere lo schermo e spostare finestre, per esempio.

Infine il messaggio CSCS converte il sistema di colori YUV usato nei parametri della televisione statunitense (NTSC) nel sistema RGB usato dai monitor dei computer. Esso si usa soprattutto quando si spedisce una immagine al terminale in formato YUV, che deve essere convertita in RGB per la visualizzazione. Questa conversione è algoritmica-

Messaggio	Significato
SET	Aggiorna un rettangolo con pixel nuovi
FILL	Riempie un rettangolo con un valore di pixel
BITMAP	Espande una bitmap per riempire un rettangolo
COPY	Copia un rettangolo da una parte del frame buffer ad un'altra
CSCS	Converte un rettangolo da colori televisivi [YUV] a RGB

Figura 5.48 I messaggi usati nel protocollo SLIM dal server ai terminali.

semplice ma computazionalmente costosa, quindi è meglio scaricare questo compito sui terminali; se il terminale non viene usato per visualizzare video, questo messaggio e la sua funzionalità non sono necessari.

L'intera idea dei client stupidi e sottili si regge o cade sulle prestazioni, misurate accuratamente da Schmidt et al. Nel loro prototipo, è stato usato un Fast Ethernet a 100 Mbps sia dal server allo switch sia dallo switch al terminale; in teoria si potrebbe usare anche un collegamento da 1 Gbps perché il segmento switch-server è localizzato nella stanza del computer centrale.

La prima misurazione riguarda i caratteri di cui si fa l'eco sullo schermo: ogni carattere digitato è mandato al server, che calcola quali pixel devono essere aggiornati per mettere il carattere sullo schermo nella posizione giusta, col font giusto e il colore giusto. Le misurazioni mostrano che occorrono 0,5 ms perché il carattere appaia sullo schermo: invece, su una workstation locale il tempo necessario al carattere per apparire sullo schermo è di 30 ms, a causa della bufferizzazione del kernel.

Il resto dei test ha misurato le prestazioni mentre utenti usano programmi applicativi interattivi medesimi, quali Photoshop della Adobe (un programma per il ritocco di fotografie), Framemaker della Adobe (un programma di editoria professionale) e Nelscape (un web browser). È stato osservato che la metà dei comandi utente richiedono un aggiornamento di meno di 10000 pixel, ossia di 30000 byte non compressi. A 100 Mbps, servono 2,4 ms per portare 10000 pixel sul cavo, più altri 2,7 ms per metterli nel frame buffer all'arrivo, per un totale di 5,1 ms (numero che può variare leggermente a seconda delle circostanze). Siccome il tempo di reazione umana è di circa 100 ms, questi aggiornamenti appaiono come istantanei, ma anche un aggiornamento più grande appare quasi istantaneo. Inoltre, usando meccanismi di compressione, più dell'85% degli aggiornamenti è al di sotto dei 30000 byte.

Gli esperimenti sono stati ripetuti con una rete a 10 Mbps, a 1 Mbps e a 128 Kbps: a 10 Mbps il sistema è quasi istantaneo e a 1 Mbps è ancora buono; ma a 128 Kbps è troppo lento per poter essere usato. Siccome connessioni a 1Mbps da casa stanno diventando rapidamente una realtà grazie alle reti a cavo televisive e ADSL (Ciclo asimmetrico di sottoscrizione digitale, Asymmetric Digital Subscriber Loop), sembra che questa tecnologia possa divenire applicabile agli utenti da casa oltre che alle aziende.

5.9 Gestione del consumo di energia

Il primo computer elettronico di uso generale, l'ENIAC, aveva 18000 valvole e consumava 140000 watt di potenza e, come risultato, produceva delle bollette elettriche non trascurabili. Dopo l'invenzione dei transistor, l'uso di energia scese enormemente, e l'industria dei calcolatori perse interesse nei requisiti di energia. Oggi, il problema del consumo di energia è tornato in auge per molti motivi e sta coinvolgendo i sistemi operativi.

Cominciamo dai PC desktop, che spesso ricevono corrente a 200 watt (di cui l'85% è usata in modo produttivo, mentre il 15% viene trasformato in calore); se 100 milioni di queste macchine vengono accese simultaneamente nel mondo, esse producono un consumo di 20000 megawatt di elettricità, corrispondente al consumo totale di 20 centrali nucleari di media grandezza. Se il consumo di energia potesse essere ridotto della metà, potremmo eliminare 10 centrali nucleari. Dal punto di vista ambientale, eliminare 10 centrali nucleari o un numero analogo di centrali a combustibile fossile è una grande conquista e quindi vale la pena di provarci.

Un altro contesto in cui il consumo di energia è una questione importante sono i com-

puter alimentati a batterie, tra cui i computer portatili, i palmari, i Webpad, ed altri. Il cuore del problema è che le batterie non possono contenere abbastanza potenza da durare a lungo, ma possono resistere qualche ora al massimo; inoltre, nonostante i grandi sforzi da parte dei produttori di batterie di computer e di industrie elettroniche, il progresso è lentissimo. Per un'industria abituata a raddoppiare le prestazioni ogni 18 mesi (legge di Moore), non avere alcun progresso suona come una violazione alle leggi della fisica, ma questa è la realtà. Di conseguenza, fare sì che i computer consumino meno energia, in modo che le batterie esistenti possano durare più a lungo, è una priorità decisamente alta. I sistemi operativi sono protagonisti di questa situazione, come vedremo in seguito.

Ci sono due approcci alla riduzione del consumo di energia; il primo è quello di fare in modo che il sistema operativo spenga parti del computer (principalmente dispositivi di I/O) quando questi non sono in uso, in quanto un dispositivo spento usa poca energia o non ne usa affatto. Il secondo è quello di far sì che programmi applicativi usino meno energia, eventualmente degradando la qualità percepita dall'utente, allo scopo di allungare la vita della batteria. Analizzeremo entrambi questi approcci, ma prima di ciò, consideriamo brevemente la progettazione hardware dal punto di vista del consumo di energia.

5.9.1 Questioni riguardanti l'hardware

Esistono due tipi principali di batterie: le batterie usa e getta e quelle ricaricabili. Le prime (le più comuni sono le celle AAA, AA e D) possono essere usate per fornire energia ad apparecchi palmari, ma non hanno abbastanza potenza per alimentare computer portatili con grandi schermi. Una batteria ricaricabile, al contrario, può immagazzinare abbastanza energia per alimentare un portatile per alcune ore. Le batterie al nickel cadmio hanno dominato in questo campo, ma hanno lasciato il posto a batterie all'idrossido di metallo e nickel, che durano più a lungo e inquinano meno l'ambiente quando vengono buttate via. Le batterie agli ioni di litio sono anche meglio, e possono venire ricaricate senza essere state precedentemente scaricate completamente, ma anche la loro capacità è molto limitata.

L'approccio generale di molti produttori di computer nei confronti della durata delle batterie è quella di progettare CPU, memoria e dispositivi di I/O in modo da prevedere più stati: acceso, attesa (sleeping), ibernazione e spento. Il dispositivo, per poter essere usato, deve essere acceso; quando un dispositivo non è necessario per un certo periodo, può essere messo in attesa, uno stato che produce una certa riduzione di consumo. Quando si pensa che non sarà necessario per un intervallo di tempo più lungo, lo si può mettere in modalità ibernazione, che riduce ulteriormente il consumo di energia; in questo caso bisogna però considerare il fatto che per svegliare un dispositivo dal modo ibernazione occorre consumare più tempo ed energia di quello che occorre per uscire dalla modalità di attesa; infine se il dispositivo è spento non fa nulla e non consuma nulla. Non tutti i dispositivi sono dotati di tutti questi stati, ma se lo sono è compito del sistema operativo di gestire la transizione da uno stato all'altro nel momento giusto.

Alcuni computer hanno due o anche tre pulsanti relativi all'alimentazione: uno di questi può mettere l'intero computer in uno stato di attesa, dal quale può essere svegliato facilmente premendo un pulsante o muovendo il mouse; un altro può mandare il computer in modo ibernazione, per uscire dal quale occorre più tempo. In entrambi i casi, questi tasti non fanno altro che spedire un segnale al sistema operativo, che completa l'operazione via software. In alcuni paesi, i dispositivi elettrici devono, per legge, avere un interruttore meccanico che interrompe il passaggio di corrente verso il dispositivo, per ragioni di sicurezza; per rispettare questa legge può essere necessario un altro tasto.

La gestione del consumo di energia solleva un certo numero di domande relative ai compiti del sistema operativo, ad esempio: quali dispositivi devono essere controllati?

Dispositivo	Li et al. (1994)	Lorch e Smith (1998)
Display	68%	39%
CPU	12%	18%
Disco fisso	20%	12%
Modem		6%
Scheda sonora		2%
Memoria	0,5%	1%
Altro		22%

Figura 5.49 Consumo di energia di varie parti di un computer portatile.

Devono essere dispositivi di tipo acceso/spento o possedere stati intermedi? Quanta energia si risparmia utilizzando modalità di basso consumo? Quanta energia si perde a riattivare il dispositivo? Occorre salvare il contesto prima di passare in basso consumo? Quanto tempo occorre prima di tornare a pieno consumo? Naturalmente le risposte a queste domande variano da dispositivo a dispositivo, e quindi il sistema operativo deve essere in grado di gestire diverse possibilità.

Molti ricercatori hanno esaminato i computer portatili per vedere dove va a finire l'energia fornita. Li et al. (1994) misurarono vari carichi di lavoro, giungendo alle conclusioni riassunte in Figura 5.49. Lorch e Smith (1998) misurarono altre macchine e giunsero alle conclusioni riassunte in Figura 5.49. Anche Weiser et al. (1994) fecero delle misurazioni, ma non pubblicarono i risultati numerici, affermando semplicemente che i principali dispositivi consumatori di energia erano i display, i dischi fissi e la CPU, in quest'ordine. Mentre i numeri non sono perfettamente concordi, forse perché avevano usato diverse marche di computer, sembra chiaro che i display, i dischi fissi e la CPU sono obiettivi ovvi per il risparmio di energia.

5.9.2 Questioni riguardanti i sistemi operativi

Il sistema operativo gioca un ruolo cruciale nella gestione dell'energia: esso controlla tutti i dispositivi e quindi deve decidere quale spegnere e quando. Se spegne un dispositivo e tale dispositivo diventa immediatamente necessario, può verificarsi un antipatico ritardo causato dall'avvio del dispositivo; d'altro canto, se aspetta troppo a lungo prima di spegnere un dispositivo, sprecherà energia per nulla.

Il trucco è quello di trovare algoritmi ed heuristiche che aiutino il sistema operativo a prendere buone decisioni, su cosa e quando spegnere; il problema è che "buone" è un concetto molto soggettivo. Un utente può considerare accettabile il fatto che, dopo che il computer è rimasto inattivo per 30 secondi, servano 2 secondi perché questo reagisca ad un tasto premuto, mentre un altro utente reagirebbe con espressioni decisamente colorite nelle stesse condizioni. In assenza di riconoscimento vocale il computer non può distinguere tra questi due tipi di utenti!

Il display

A questo punto diamo un'occhiata ai principali centri di consumo di energia e vediamo cosa si può fare per ognuno di essi. L'elemento principale in qualunque bilancio di

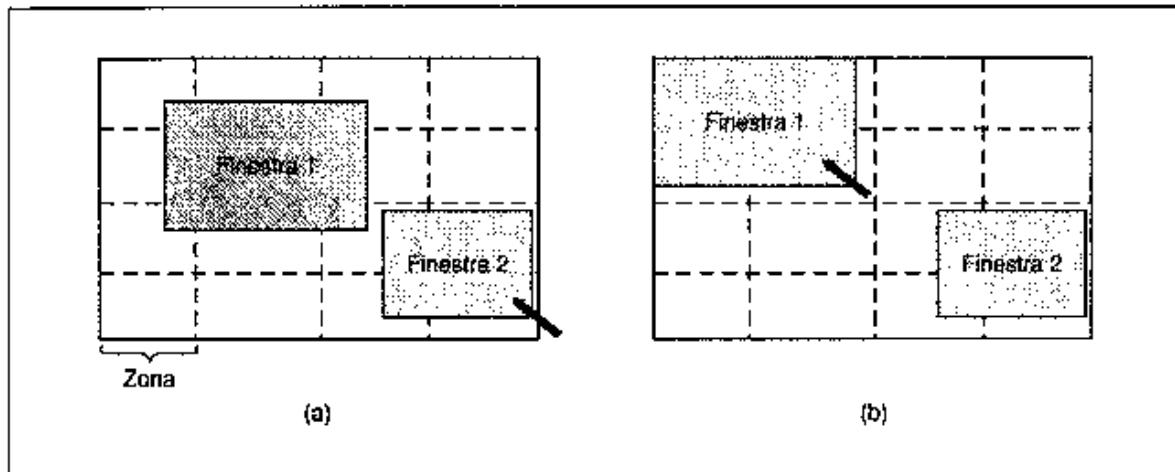


Figura 5.50 L'uso di zone per la retroilluminazione del display. [a] Quando la finestra 2 viene selezionata non c'è bisogno di spostarla. [b] Quando si seleziona la finestra 1, questa viene mossa per ridurre il numero di zone illuminate.

energia è il display: per ottenere un'immagine grande e definita, lo schermo deve essere retroilluminato, e questo richiede una certa quantità di energia. Molti sistemi operativi cercano di risparmiare energia spegnendo il display quando è rimasto inattivo per qualche minuto; spesso l'utente può scegliere la lunghezza degli intervalli dopo i quali il display viene spento, gestendo personalmente il compromesso tra un display che si disattiva ogni pochi minuti ed una batteria che si consuma in fretta (e non è detto che l'utente voglia la responsabilità di questa scelta). Lo spegnimento del display è in realtà uno stato di attesa, dal quale esso può venire rigenerato (tramite la RAM video) quasi istantaneamente premendo un tasto o muovendo il mouse.

Un possibile miglioramento è stato proposto da Flinn e Satyanarayanan (1999), i quali hanno suggerito di dividere il display in un certo numero di zone che possano essere alimentate in modo indipendente. In Figura 5.50 abbiamo rappresentato 16 zone usando le linee tratteggiate come delimitazione. Quando il cursore è sulla finestra 2, come mostrato in Figura 5.20(a), solo le quattro zone nell'angolo in basso a destra devono venire accese, mentre le altre 12 possono rimanere buie, producendo un risparmio energetico di 3/4 di potenza.

Quando l'utente muove il cursore sulla finestra 1, le zone relative alla finestra 2 possono venire spente, mentre le zone dietro alla finestra 1 vengono accese; comunque, siccome la finestra 1 tocca 9 zone, il consumo aumenta. Se il gestore delle finestre si accorgesse di questo, può muovere automaticamente la finestra 1 in modo da ricoprire 4 sole zone, come mostrato in Figura 5.50(b). Per ottenere questa riduzione dai 9/16 del consumo totale ai 4/16, il gestore di finestre deve essere in grado di analizzare questioni relative alla gestione dell'energia, o accettare istruzioni da qualche parte del sistema che sia in grado di farlo. In modo ancora più sofisticato si potrebbe illuminare solo parzialmente una zona che non è completamente piena (ad esempio, una finestra che contiene linee di testo corte potrebbe essere lasciata al buio sul lato destro).

Il disco fisso

Un altro grande colpevole è il disco fisso, che richiede una grande quantità di energia per tenere alta la velocità di rotazione, anche se non vi sono accessi al disco. Molti com-

puter, specialmente i portatili, rallentano la velocità di rotazione del disco soprattutto dopo un certo numero di minuti di inattività, ed aumentano la velocità nel caso il disco venga nuovamente richiesto. Sfortunatamente, fermare un disco significa ibernarlo piuttosto che metterlo in attesa, in quanto occorrono alcuni secondi per farlo ripartire, che risultano in ritardi facilmente rilevabili dall'utente.

Inoltre, il fatto di far ripartire il disco produce un consumo di altra energia e, di conseguenza, ogni disco ha un tempo caratteristico, T_d , che rappresenta il punto limite per un vero guadagno nei consumi, solitamente nell'intorno da 5 a 15 secondi: supponiamo di aver stimato che l'accesso successivo al disco sia al tempo t nel futuro; se $t < T_d$, occorre meno energia a lasciare andare il disco piuttosto che rallentarlo e poi farlo ripartire, mentre se $t > T_d$, allora risulta conveniente rallentare il disco. Se si potessero fare delle previsioni affidabili (ad esempio, basandosi sulle strutture degli accessi precedenti), il sistema operativo potrebbe fare buone previsioni su quando disattivare il disco fisso, e quindi risparmiare molta energia. In pratica, la maggior parte dei sistemi sono conservativi, e rallentano il disco solo dopo alcuni minuti di inattività.

Un altro modo di risparmiare l'energia del disco è quello di avere una grande cache di disco nella RAM; se un blocco richiesto è già nella cache, non occorre far ripartire un eventuale disco inattivo; in modo simile, se le scritture su disco possono essere inserite nella cache, un disco inattivo non deve essere riavviato per completare la scrittura. Il disco può rimanere spento finché la cache non è piena o finché non si richiede un dato non presente.

Un altro modo ancora per evitare avvii non necessari del disco è fare in modo che il sistema operativo informi i programmi in esecuzione sullo stato del disco, tramite segnali o messaggi. Alcuni programmi hanno delle scritture opzionali che possono essere saltate o ritardate; per esempio un elaboratore di testi (word processor) può essere impostato per scrivere su disco le modifiche al file ogni qualche minuto; se l'elaboratore di testi sa che il disco è spento nel momento in cui salverebbe le modifiche, può ritardare la scrittura a quando il disco viene riaccesso, o comunque dopo un ulteriore periodo di tempo.

La CPU

Anche la CPU può essere gestita in modo da risparmiare energia; la CPU di un portatile può essere messa in modo di attesa tramite software, riducendo il consumo di energia quasi a zero: l'unica cosa che può fare quando si trova in questo stato è svegliarsi quando si verifica un'interruzione. Quindi ogniqualvolta la CPU sia inattiva, o in attesa di I/O o perché non c'è alcun lavoro da fare, si mette in attesa.

Su molti computer esiste una correlazione tra il voltaggio della CPU, i cicli di clock e l'uso dell'energia. Il voltaggio di CPU, spesso, può venire ridotto via software, la qual cosa produce un risparmio di energia, ma riduce anche il ciclo di clock (in modo quasi lineare). Siccome l'energia consumata è proporzionale al quadrato del voltaggio, dimezzare il voltaggio rallenta la CPU della metà, ma riduce ad 1/4 il suo consumo.

Questa proprietà può essere sfruttata per programmi con scadenze ben definite, quali i visualizzatori di dati multimediali che devono decomprimere e visualizzare un fotogramma ogni 40 ms, ma diventano inattivi se riescono a completare queste operazioni in un tempo minore. Supponiamo che la CPU usi x Joule girando per 40 ms a piena potenza, e $x/4$ a mezza potenza; se un visualizzatore multimediale è in grado di decomprimere un fotogramma e visualizzarlo in 20 ms, il sistema operativo può andare a piena potenza per 20 ms e fermarsi per 20 ms, per un consumo di energia totale di $x/2$ joule. Alternativamente, può andare a potenza dimezzata e arrivare alla scadenza appena in tempo, ma usare $x/4$ di potenza. Un confronto tra esecuzioni a piena potenza per un certo inter-

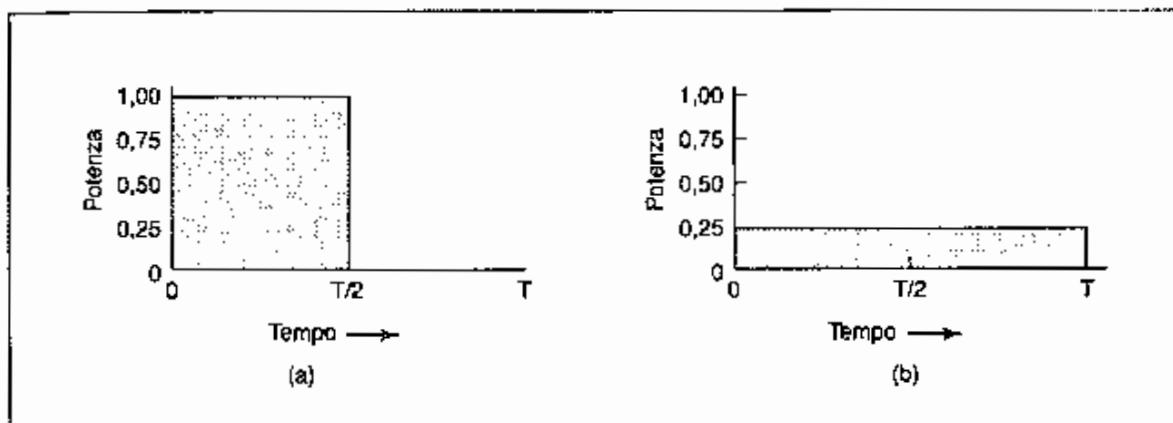


Figura 5.51 (a) Esecuzione a pieno tempo di clock. (b) Dimezzare il voltaggio dimezza il ciclo di clock e divide per quattro il consumo.

vallo di tempo, e a potenza dimezzata e ad un quarto di consumo per un tempo doppio, è mostrato in Figura 5.51: in entrambi i casi viene eseguito lo stesso lavoro, ma in Figura 5.51(b) si consuma metà dell'energia.

In modo simile, se un utente sta battendo 1 carattere al secondo, ma il lavoro necessario ad elaborare un carattere richiede 100 ms, è meglio se il sistema operativo si accorge che ci sono lunghe latenze e rallenta la CPU di un fattore 10. In altre parole, le esecuzioni lente producono maggior risparmio energetico delle esecuzioni veloci.

La memoria

Esistono due modi possibili di risparmiare energia tramite la memoria: primo, la cache può essere liberata e poi spenta e, in ogni caso, può in seguito venire ricaricata dalla memoria centrale senza perdita di informazioni; quest'ultima operazione può essere fatta in modo dinamico e veloce, e quindi spegnere la cache significa effettivamente metterla in uno stato di attesa.

Un'opzione molto più drastica è quella di scrivere il contenuto della memoria centrale sul disco e poi spegnere la memoria centrale stessa. Questo approccio equivale ad un'ibernazione, in quanto quasi tutta l'energia può essere risparmiata, a spese del tempo di riavvio, specialmente se anche il disco viene spento. Quando si disattiva la memoria, o si spegne anche la CPU, o essa deve eseguire codice in ROM: se viene spenta, l'interruzione che la sveglia deve causare un salto nella ROM in modo da ricaricare la memoria prima di usarla. Nonostante tutto l'overhead, spegnere la memoria per lunghi periodi di tempo (ad esempio, ore) può essere una buona idea, nel caso farla ripartire in pochi secondi sia considerato meglio che far ripartire il sistema operativo del disco, cosa che spesso richiede più di un minuto.

Comunicazione wireless

Diventa sempre più comune per i computer portatili avere connessioni wireless (senza fili) con il mondo esterno (ad esempio, Internet). I trasmettitori e ricevitori radio richiesti sono spesso delle sanguisughe di energia di prima classe. In particolare, se il ricevitore radio è sempre acceso in attesa dell'arrivo di e-mail, la batteria può prosciugarsi mol-

to velocemente. D'altro canto, se la radio viene spenta dopo, ad esempio, un minuto di inattività, si possono perdere messaggi in ingresso, cosa che chiaramente non è desiderabile.

Una soluzione efficiente a questo problema è stata proposta da Kravets e Krishnan (1998). Il cuore della loro soluzione sfrutta il fatto che i computer portatili comunicano con stazioni fisse che hanno grandi capacità di memoria e grandi dischi e nessun limite di energia; in base alla loro proposta, il portatile invia un messaggio alla stazione base nel momento in cui sta per spegnere la radio. Da quel momento in poi, la stazione immagazzina i messaggi in arrivo sul suo disco; quando il computer portatile riaccende la radio, avverte la stazione, la quale può inviarli tutti i messaggi accumulati.

I messaggi in uscita generati mentre la radio è spenta vengono memorizzati sul portatile e quando il buffer minaccia di riempirsi la radio viene accesa e la coda di messaggi viene trasmessa alla stazione.

Quando si dovrebbe spegnere la radio? Una possibilità è lasciare questa decisione all'utente o al programma applicativo; un'altra è di spegnerla dopo un certo numero di secondi di inattività. Quando la si dovrebbe riaccendere? Ancora una volta, l'utente o il programma possono decidere, oppure la si può riaccendere periodicamente e controllare il traffico e la coda di messaggi da spedire: naturalmente la si dovrebbe accendere quando il buffer di messaggi in uscita è quasi pieno, ma esistono molte altre euristiche possibili.

Gestione termica

Una questione leggermente diversa, ma sempre legata al consumo di energia, è la questione termica. Le CPU moderne diventano estremamente calde a causa delle loro alte velocità; i computer fissi normalmente hanno delle ventole elettriche interne che permettono di spingere l'aria calda al di fuori dello chassis e, siccome al momento il consumo di energia non è un problema di punta nel caso di computer fissi, di solito la ventola è continuamente accesa.

Con i computer portatili la situazione è diversa; i sistemi operativi devono controllare continuamente la temperatura e, quando questa diventa prossima al massimo consentito, il sistema operativo ha una scelta: o accende la ventola, che fa rumore e consuma energia, o riduce il consumo di energia riducendo l'illuminazione dello schermo, rallentando la CPU, facendo scelte più drastiche relativamente al rallentamento del disco e così via.

Qualche aiuto da parte dell'utente può essere una guida utile; ad esempio, un utente può dire in anticipo di non sopportare il rumore della ventola, quindi il sistema operativo sa che deve ridurre il consumo di energia piuttosto che azionare la ventola.

Gestione della batteria

C'era una volta una batteria, che forniva energia finché non era scarica e a quel punto si fermava. Questo non è più vero: oggi i computer portatili usano batterie intelligenti, che comunicano col sistema operativo; su richiesta esse possono fornire informazioni quali il voltaggio massimo, il voltaggio corrente, la carica massima, la carica corrente, la percentuale massima di scarica, quella corrente e altre cose. La maggior parte dei portatili hanno programmi che possono interrogare la batteria e visualizzare tutti questi parametri. Le batterie intelligenti possono anche imparare a cambiare parametri operativi sotto il controllo del sistema operativo.

Alcuni portatili hanno più di una batteria; quando il sistema operativo rileva che una batteria è quasi scarica, deve organizzare un passaggio indolore all'altra batteria, senza causare malfunzionamenti o irregolarità durante la transizione. Quando l'ultima batteria

è in via di esaurimento, sta al sistema operativo avvertire l'utente e causare uno spegnimento metodico, ad esempio controllando che il file system non sia corrotto.

Interfaccia driver

Il sistema Windows ha un complesso meccanismo di gestione dell'energia chiamato ACPI (*configurazione avanzata e interfaccia di alimentazione*, Advanced Configuration and Power Interface): esso può inviare comandi ad ogni driver conforme a tale interfaccia, chiedendo informazioni sulla capacità dei suoi drive e sul loro stato corrente. Questa caratteristica è molto importante se combinata al plug and play, dato che all'avvio il sistema operativo non sa quanti dispositivi siano presenti e, a maggior ragione, non conosce le loro proprietà in termini di consumo e gestione di energia.

Esso può anche inviare messaggi ai drive, istruendoli a diminuire il loro consumi di energia (in base alle capacità che ha imparato in precedenza, ovviamente); si ha anche un po' di traffico nell'altra direzione, in particolare, quando un dispositivo quale la tastiera o il mouse rilevano qualche attività dopo un periodo di inattività, avvertono il sistema di tornare ad uno stato di normalità (o quasi).

5.9.3 Operazioni degradate

Fino ad ora abbiamo analizzato i modi in cui il sistema operativo può ridurre l'uso di energia da parte di vari dispositivi; esiste però un altro approccio: impostare i programmi in modo che usino meno energia, anche se questo significa fornire un servizio peggiore all'utente (meglio un servizio peggiore che nessun servizio, se la batteria muore e le luci si spengono). Tipicamente, questa informazione viene passata al programma quando la carica della batteria scende al di sotto di una determinata soglia e a questo punto sta al programma decidere se degradare le prestazioni e allungare la vita della batteria, o mantenere le prestazioni e rischiare l'esaurimento di energia.

Una delle domande che vengono in mente a questo punto è: come può un programma degradare le prestazioni allo scopo di risparmiare energia? La risposta è stata studiata da Flinn e Satyanarayanan (1999), i quali fornirono quattro esempi di come prestazioni degradate possono far risparmiare energia. A questo punto analizziamo i loro esempi.

In questo studio, l'informazione viene presentata all'utente in varie forme e in assenza di degradazione l'utente riceve la miglior qualità di informazione; in presenza di degradazione, la fedeltà (accuratezza) dell'informazione presentata all'utente è peggiore di come avrebbe potuto essere; tra un secondo vedremo alcuni esempi di questo.

Allo scopo di misurare l'uso di energia, Flinn e Satyanarayanan idearono uno strumento software chiamato PowerScope, che fornisce il profilo di uso di energia di un programma. Per usarlo, il computer deve essere collegato ad una fonte di energia esterna tramite un multimetro digitale controllato dal software; il software di controllo può leggere il numero di milliampere che entrano dalla fonte di energia, e determinare in modo istantaneo l'energia consumata dal computer. PowerScope campiona periodicamente il contatore di programma (program counter) e l'uso di energia e scrive questi dati in un file; dopo che il programma ha terminato si analizza il file, e si forniscono i consumi di energia di ogni procedura: queste misure formano la base delle loro osservazioni. Sono state usate anche misure hardware di risparmio di energia, come base per il confronto delle misure di risparmio basate su prestazioni ritardate.

Il primo programma misurato era un lettore di video. In modo non degradato, visualizza 30 fotogrammi al secondo a piena risoluzione e a colori. Una prima forma di degradazione è abbandonare il colore, e visualizzare l'informazione video in bianco e nero;

un'altra forma di degradazione è quella di ridurre la frequenza dei fotogrammi (frame rate), producendo scatti e dando al film un aspetto traballante. Un'altra forma di degradazione ancora è quella di ridurre il numero di pixel per fotogramma in entrambe le direzioni, o diminuendo la risoluzione spaziale o rendendo l'immagine più piccola. Misure di questo tipo producono un risparmio di energia del 30%.

Il secondo programma era un riconoscitore vocale, che campionava il segnale ricevuto attraverso il microfono e costruiva una forma d'onda. Questa onda può essere analizzata sul computer portatile o inviata tramite radio ad un computer fisso; questo produce un risparmio di energia di CPU, ma produce un consumo di energia di trasmissione radio. La degradazione è stata ottenuta usando un vocabolario più piccolo e un modello acustico più semplice; si è ottenuto un risparmio energetico del 35%.

L'esempio successivo riguarda un visualizzatore di mappe che ricuperà la mappa via radio. La degradazione consiste nel tagliare la mappa in dimensioni più piccole o dicendo al server remoto di omettere le strade minori, riducendo il numero di bit durante la trasmissione. Anche in questo caso si è risparmiato il 35%.

Il quarto esperimento riguardava la trasmissione di immagini JPEG ad un Web browser. Lo standard JPEG permette vari algoritmi, in cui si seleziona la qualità dell'immagine a scapito della dimensione del file. In questo caso il guadagno è stato solo intorno al 9%. Ciononostante, gli esperimenti mostrano che accettando qualche degradazione nella qualità dei risultati, l'utente può utilizzare più a lungo la batteria.

5.10 Ricerca nel campo dell'I/O

C'è una certa quantità di ricerca nel campo dell'input/output, ma è principalmente focalizzata su dispositivi specifici piuttosto che sull'I/O in generale; spesso lo scopo è di migliorare le prestazioni, in un modo o nell'altro.

I sistemi di dischi sono un esempio classico. I primi algoritmi di schedulazione del movimento dei bracci dei dischi usano un modello di disco che non è più utilizzabile, quindi Worthington et al. (1994) hanno analizzato modelli che corrispondono ai nuovi dischi. RAID è un argomento caldo e molti ricercatori ne esaminano diversi aspetti: Alvarez et al. (1997) hanno considerato miglioramenti della tolleranza ai guasti, così come Blaum et al. (1994). Cao et al. (1994) hanno esaminato l'idea di avere un controllore parallelo su un sistema RAID, mentre Wilkes et al. (1996) hanno descritto un RAID avanzato costruito alla HP. La presenza di drive multipli richiede una buona schedulazione parallela, quindi anche questo è un argomento di ricerca diffuso (Chen e Towsley, 1996; Kalalahalla e Barman, 1999). Lumb et al. (2000) considerano l'idea di utilizzare il tempo di attesa dopo un posizionamento, ma prima che il settore richiesto ruoti sotto la testina, per precaricare dati. Meglio che usare il tempo di latenza nella rotazione per fare lavoro utile, è l'idea di eliminare completamente la rotazione, e usare dispositivi di memorizzazione microelettromeccanici allo stato solido (Griffin et al. 2000; Carley et al., 2000), o memorizzazione olografica (Orlov, 2000). Un'altra nuova tecnologia che vale la pena di controllare è la memorizzazione magneto-ottica (McDaniel, 2000).

I terminali SLIM forniscono una versione moderna dei vecchi sistemi a condivisione di tempo, con tutto il calcolo fatto in modo centralizzato e l'utente che lavora da un terminale che gestisce direttamente solo display, mouse e tastiera (Schmidt et al., 1999). La differenza principale con i vecchi sistemi a condivisione del tempo è che invece di collegare il terminale al computer tramite un modem a 9600 bps, si usa un collegamento Ethernet a 10 Mbps, che fornisce abbastanza banda per un'interfaccia grafica completa dal lato dell'utente.

Le GUI sono al momento abbastanza standard, ma c'è ancora molto lavoro in corso in quest'area, ad esempio nell'ambito di input vocale (Malkewitz, 1988; Manaris e Harkreader, 1998; Slaughter et al., 1998; VanBuskirk e LaLomia, 1995). Anche la struttura interna di una GUI è un argomento di ricerca (Taylor et al., 1995).

Dato il grande numero di informatici che usano portatili e dato il tempo microscopico di durata della loro batteria, non dovrebbe sorprendere che vi sia un interesse così grande nel cercare di risparmiare energia tramite tecniche software (Ellis, 1999; Flin e Satyanarayanan, 1999; Kravets e Krishnan, 1998; Lebeck et al., 2000; Lorch e Smith, 1996; Lu et al., 1999).

5.11 Sommario

L'I/O è un argomento molto importante, anche se spesso viene un po' trascurato; in ogni sistema operativo una porzione sostanziale del codice è dedicata all'I/O. L'I/O può essere effettuato in tre modi: prima di tutto c'è l'I/O programmato, in cui la CPU principale mette in input o in output ogni singolo byte o parola e si mette in loop aspettando di poter muovere il byte successivo; il secondo modo è l'I/O guidato dalle interruzioni, in cui la CPU inizia un trasferimento di carattere o parola, e poi si dedica a qualcosa' altro finché un'interruzione la avverte del completamento dell'operazione di I/O. Infine c'è il DMA, nel quale un circuito separato gestisce il completo trasferimento di un blocco di dati, e invia un'unica interruzione quando l'intero trasferimento è stato effettuato.

L'I/O può essere strutturato in quattro livelli: le procedure di servizio delle interruzioni, i driver di dispositivo, il software di I/O indipendente dal dispositivo, e le librerie di I/O e di spool che si trovano nello spazio utente. I driver di dispositivo gestiscono i dettagli relativi all'esecuzione dei dispositivi, e forniscono interfacce uniformi al resto del sistema operativo. Il software di I/O indipendente dal dispositivo si occupa di problemi quali la bufferizzazione e la comunicazione di errori.

Esistono vari tipi di disco, tra cui i dischi magnetici, i RAID e vari tipi di dischi ottici. Gli algoritmi di schedulazione del braccio dei dischi sono spesso usati per migliorare le prestazioni del disco, ma la presenza di geometrie virtuali complica il problema; accoppiando due dischi, si può ottenere un mezzo di memorizzazione stabile, con certe proprietà utili.

I clock sono usati per tenere traccia del tempo vero e proprio, limitare la durata di un processo, gestire timer di controllo e tenere la contabilità.

I terminali orientati a carattere hanno una varietà di problemi relativi ai caratteri speciali che possono essere dati in input e sequenze di escape speciali in output; l'input può essere in modo raw o modo cooked, a seconda di quanto il programma voglia controllare l'input. Le sequenze di escape in output controllano i movimenti del cursore, e permettono l'inserimento o la cancellazione di testo sullo schermo.

Molti personal computer usano GUI per il loro output; queste sono basate sul paradigma WIMP: finestre (windows), icone (icons), menu (menus) e dispositivi di puntaimento (pointing devices). I programmi basati su GUI sono solitamente guidati dagli eventi, cioè gli eventi provenienti da tastiera, mouse o altro sono inviati al programma per l'elaborazione.

Esistono diversi terminali di rete, i più popolari usano X, un sistema sofisticato che può essere usato per costruire diverse GUI. Un'alternativa a X Windows è un'interfaccia di basso livello che semplicemente invia dei pixel attraverso la rete; esperimenti fatti col terminale SLIM mostrano che questa tecnica funziona sorprendentemente bene.

Infine, la gestione della corrente è una questione fondamentale per i computer porta-

tili in quanto la vita delle batterie è limitata. Il sistema operativo può usare svariate tecniche per ridurre il consumo di energia, e i programmi possono aiutare in ciò sacrificando la qualità per favorire la durata delle batterie.

PROBLEMI

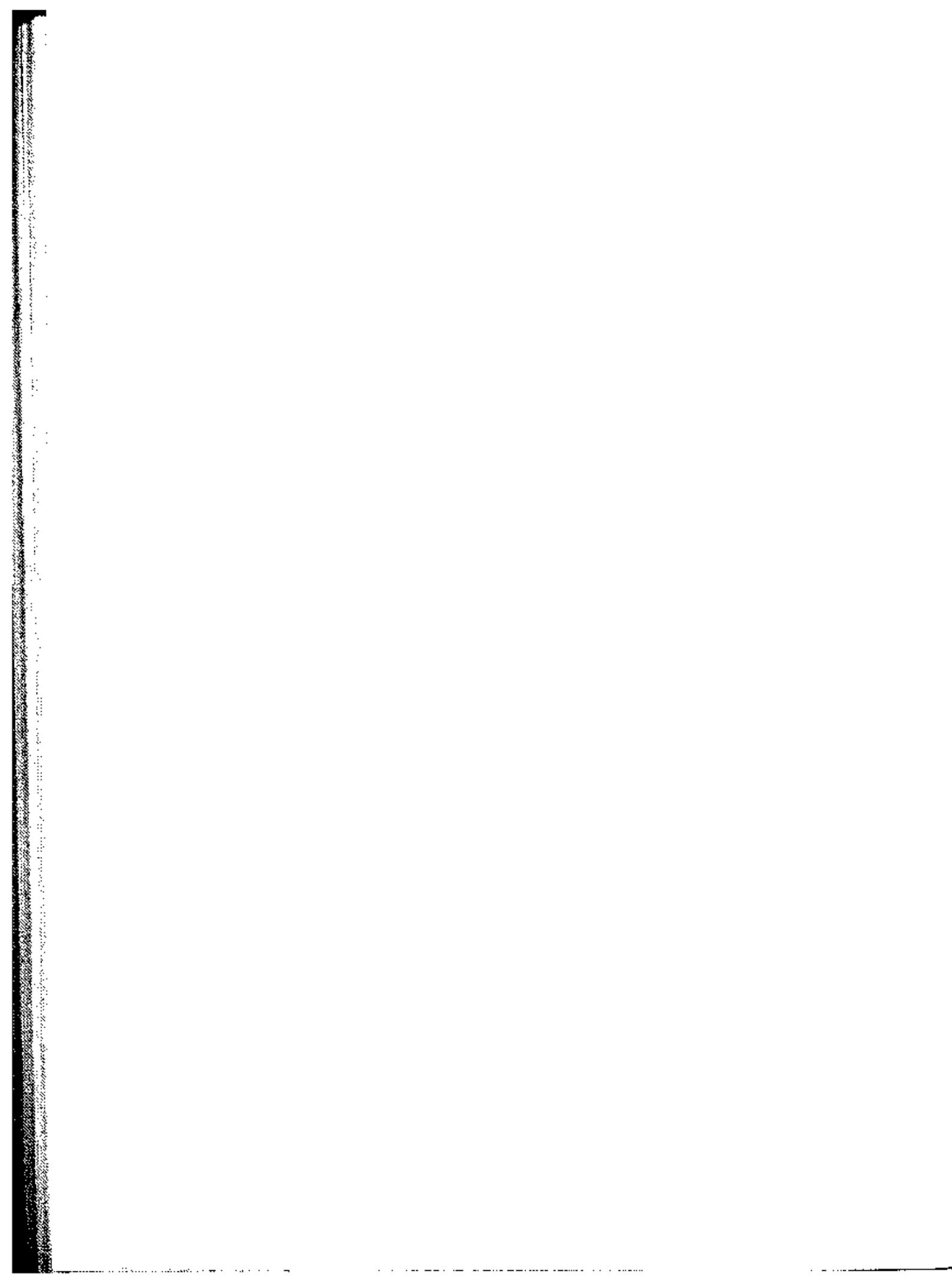
1. Gli sviluppi nella tecnologia dei circuiti integrati hanno reso possibile mettere un intero controllore, con tutta la logica di accesso al bus, su un circuito economico. In che modo questo influenza lo schema di Figura 1.5?
2. Data la lista di velocità di Figura 5.1, è possibile scannerizzare documenti con uno scanner e copiarli in un disco EIDE collegato ad un bus ISA, il tutto a velocità massima? Motiva la tua risposta.
3. La Figura 5.3(b) mostra un modo di concepire l'I/O mappato in memoria anche in presenza di bus separati per la memoria e i dispositivi di I/O, ossia quello di tentare prima il bus di memoria e, se fallisce, usare il bus di I/O. Uno studente di informatica intelligente ha pensato ad un miglioramento di questa idea: provare entrambi i bus in parallelo, per velocizzare l'accesso ai dispositivi di I/O. Cosa ne pensi di questa idea?
4. Un controllore DMA ha quattro canali ed è in grado di richiedere una parola di 32 bit ogni 100 ns. Una risposta richiede la stessa quantità di tempo. Quanto deve essere veloce il bus per evitare di diventare un collo di bottiglia?
5. Supponi che un computer possa leggere o scrivere una parola in memoria in 10 ns. Supponi inoltre che quando si verifica un'interruzione, tutti i 32 registri di CPU, più il contatore di programma (program counter) più il PSW siano infilati nella pila. Qual è il massimo numero di interruzioni per secondo che questa macchina può elaborare?
6. In Figura 5.8(b), non viene inviato l'acknowledgement di un'interruzione finché il carattere successivo non è stato dato in output sulla stampante. Si sarebbe potuto dare l'acknowledgement all'inizio della procedura di servizio dell'interruzione? Se sì, presenta una ragione per farlo alla fine, così come fatto nel testo. Se no, perché no?
7. Un computer ha una pipeline a tre passi, come mostrato in Figura 1.6(a). Per ogni ciclo di clock, si cerca in memoria l'istruzione puntata dal PC, la si mette nella pipeline e si incrementa il PC. Ogni istruzione occupa esattamente una parola di memoria. Le istruzioni che si trovano già nella pipeline vengono avanzate di un passo. Quando si verifica un'interruzione, il PC corrente viene messo sullo stack e gli viene assegnato l'indirizzo del gestore di interruzioni. Poi la pipeline viene spostata di un passo verso destra e la prima istruzione del gestore di interruzioni viene caricata nella pipeline. Questa macchina ha delle interruzioni precise? Motiva la tua risposta.
8. Una pagina di testo stampata tipicamente contiene 50 linee di 80 caratteri l'una. Immagina che una certa stampante possa stampare 6 pagine al minuto e che il tempo per scrivere un carattere sul registro di uscita della stampante sia così breve da poter essere ignorato. Ha senso utilizzare un I/O guidato dalle interruzioni per questa stampante se ogni carattere stampato richiede un'interruzione che può essere servita in $50\mu\text{s}$?

9. Cosa si intende per "indipendenza dal dispositivo"?
10. In quale dei quattro livelli di software di I/O si trovano le seguenti cose:
 - (a) Calcolo della traccia, del settore e della testina per la lettura da disco.
 - (b) Scrittura di comandi sui registri di dispositivo.
 - (c) Controllo dei diritti di un utente all'uso di un dispositivo.
 - (d) Conversione da interi binari a ASCII prima di una stampa.
11. In base ai dati di Figura 5.17, qual è la velocità di trasferimento tra disco e controllore per un dischetto e un disco fisso? Come confrontare questi dati con un modem a 56 Kbps e una Fast Ethernet a 100 Mbps rispettivamente?
12. Una rete locale viene usata nella seguente maniera. L'utente invia una chiamata al sistema per scrivere sulla rete. Il sistema operativo copia i dati in un buffer del nucleo e poi li copia sulla scheda del controllore della rete. Quando tutti i byte sono stati trasferiti in maniera sicura sulla scheda del controllore, essi vengono spediti sulla rete a 10 Mbit al secondo. Il controllore della rete che li riceve memorizza un bit dopo un microsecondo dalla spedizione. Quando tutti i bit sono arrivati, il processore del destinatario viene interrotto ed il nucleo copia il pacchetto appena arrivato in un suo buffer per poterlo controllare. Una volta identificato l'utente destinatario, i dati vengono copiati nel suo spazio di indirizzamento. Se supponiamo che ogni interruzione e la procedura che gli è associata siano servite in un millisecondo, che i pacchetti siano di 1024 byte (ignorare le intestazioni), e che per copiare un byte ci voglia 1 ms, qual è la massima velocità alla quale un processo può inviare dati verso un altro processo? Supponi che il mittente sia bloccato finché il procedimento non è completato dal lato destinatario e non ne riceve un acknowledgement. Per semplicità, supponi che il tempo necessario per ricevere l'acknowledgement sia così piccolo da poter essere ignorato.
13. Perché, normalmente, prima della stampa viene fatto lo spool su disco dei file per la stampante?
14. Che pendenza di cilindro è necessaria per un disco a 7200 giri al minuto, con un tempo di posizionamento da traccia a traccia di 1 ms? Il disco ha 200 settori di 512 byte l'uno su ogni traccia.
15. Calcolare la massima velocità di dati in MB/s per il disco descritto nel problema precedente.
16. Il livello 3 di RAID è in grado di correggere errori a singolo bit usando solo un drive di parità. Qual è la ragione dell'esistenza di RAID livello 2? Dopo tutto, può solo correggere un errore e richiede più drive.
17. Un RAID può fallire se si verifica il crash di due o più dei suoi drive in un breve intervallo di tempo. Supponi che la probabilità di crash di un drive in una certa ora sia p . Qual è la probabilità che un RAID di k dischi possa fallire nell'ora data?
18. Perché i dispositivi di memorizzazione ottica hanno potenzialità di densità di dati più alte dei dispositivi di memorizzazione magnetica? *Nota:* il problema richiede conoscenze di base della fisica studiata nelle scuole superiori e di come vengano generati i campi magnetici.
19. Se un controllore del disco scrive i byte che riceve dal disco nella memoria alla stessa velocità con cui li riceve, senza memorizzarli in un buffer interno, l'interleaving è ancora utile? Discutine.

20. Un dischetto ha l'interleaving doppio come mostrato dalla Fig. 5.26(c). Ha 8 settori di 512 caratteri per traccia e una frequenza di rotazione di 300 rpm. Quanto tempo impiega per leggere, nell'ordine, tutti i settori di una traccia, supponendo che il braccio delle testine sia già posizionato sulla traccia, e che ci voglia mezza rotazione perché il settore zero passi sotto la testina? Quant'è il tasso di trasferimento dei dati? Ripeti l'esercizio per un disco con le stesse caratteristiche ma senza l'interleaving. Di quanto viene degradato il tasso di trasferimento dei dati a causa dell'interleaving?
21. Se un disco ha doppio interleaving, c'è anche bisogno di pendenza del disco per evitare di mancare i dati quando si effettua un posizionamento da traccia a traccia? Discuti la tua risposta.
22. Un produttore di dischi ha due dischi da 5,25 pollici ognuno costituito da 10000 cilindri. Il più nuovo ha una densità di registrazione lineare che è il doppio di quello vecchio. Quali proprietà del disco sono migliori sul drive nuovo e quali sono uguali?
23. Un produttore di computer decide di riprogettare la tabella delle partizioni di un disco fisso di un Pentium, in modo da fornire più di quattro partizioni. Quali conseguenze può avere questo cambiamento?
24. Al driver del disco arrivano, nell'ordine, richieste per i cilindri 10, 22, 20, 2, 40, 6 e 38. Uno spostamento da una traccia a quella adiacente richiede 6 millisecondi. Quanto tempo è necessario per servire le richieste con:
- La politica First-Come First-Served.
 - La politica che sceglie per primo il cilindro più vicino.
 - L'algoritmo dell'ascensore (iniziando verso l'alto).
- Supporre, in tutti i casi, che il braccio si trovi inizialmente posizionato sul cilindro 20.
25. Un venditore di PC in visita presso una Università a Sud Ovest di Amsterdam, durante il suo giro di vendite, afferma che la sua compagnia ha dedicato un ampio sforzo a rendere molto veloce la loro versione di UNIX. Come esempio, egli fa notare che il loro driver del disco usa l'algoritmo dell'ascensore e accoda le richieste relative ad uno stesso cilindro in ordine di settore. Uno studente, Harry Hacker, rimane colpito da queste affermazioni, si compra un PC, lo porta a casa e scrive un programma che legge in maniera casuale 10000 blocchi sparsi sul disco. Con sua sorpresa scopre che ha misurato una prestazione identica a quella che si sarebbe aspettato se fosse stata usata la politica First-Come First-Served. Il venditore ha mentito?
26. Nella discussione relativa alla memorizzazione stabile tramite RAM non volatile, il seguente punto è stato ignorato. Cosa succede se la scrittura stabile viene completata ma si verifica un crash prima che il sistema operativo possa scrivere un numero di blocco non valido sulla RAM non volatile? Questa condizione di corsa critica rovina l'astrazione della memorizzazione stabile? Spiega le tue risposte.
27. Il gestore di interruzioni del clock di certi computer richiede 2 ms (compreso l'overhead di cambio del processo) per un colpo di clock. Il clock viene eseguito a 60 Hz. Quale percentuale della CPU è dedicata al clock?
28. Molte versioni di UNIX usano un intero senza segno di 32 bit per tenere traccia del tempo come numero di secondi rispetto al tempo origine. Quando si azzereranno questi sistemi (anno e mese)? Ti aspetti che questo succederà veramente?

29. Alcuni computer hanno bisogno di un grande numero di linee RS-232, ad esempio i server o Internet provider. Per questo motivo, esistono schede plug-in con linee RS-232 multiple. Supponi che queste schede contengano un processore che deve campionare ogni linea in entrata a 8 volte la frequenza di baud per controllare se il bit in entrata è 0 o 1. Supponi inoltre che questa campionatura richiede 1 μ s. Per linee a 28800 bps che operano a 3200 baud, quante linee può supportare il processore? Nota: la frequenza di baud di una linea è il numero di cambiamenti del segnale al secondo. Una linea a 3200 baud può supportare 28800 bps se ogni intervallo del segnale codifica 9 bit, usando diverse ampiezze, frequenze e fasi. Incidentalmente, i modem a 56K non usano RS-232 e quindi non sono un buon esempio di temporizzazione RS-232.
30. Perché i terminali RS-232 sono gestiti tramite interruzioni, mentre quelli mappati in memoria non lo sono?
31. Considera le prestazioni di un modem a 56 Kbps. Il driver mette in output un carattere e si blocca. Quando il carattere è stato emesso, si verifica un'interruzione e viene inviato un messaggio al driver bloccato. Se il tempo per passare il messaggio, mettere il carattere in output e bloccare il driver è di 100 μ s, quale percentuale del tempo di CPU viene consumata dalla gestione del modem? Supponi che ogni carattere abbia un bit di inizio e uno di fine e che quindi sia di 10 bit.
32. Un terminale grafico contiene 1280 per 960 pixel. Per far scorrere il contenuto di una finestra, il processore (o il controllore) deve spostare tutte le linee di testo verso l'alto copiando i bit che le compongono da una zona della RAM video all'altra. Se una certa finestra è alta 66 linee e larga 80 caratteri (5280 caratteri, in tutto) e l'area in cui si disegna un carattere è alta 16 pixel e larga 8, quanto tempo ci vuole per far scorrere la finestra, se occorrono 50 ns per spostare un singolo byte? Se tutte le linee sono lunghe 80 caratteri, qual è la velocità equivalente in baud per il terminale? Stampare un carattere sullo schermo è un'operazione che impiega 5 μ s. Quante linee per secondo possono venire visualizzate?
33. Dopo aver ricevuto un carattere DEL (SIGINT), il driver del display elimina tutto l'output che al momento è in coda per la visualizzazione. Perché?
34. Un utente ad un terminale RS-232 emette un comando di editor per la cancellazione di una parola della linea 5, che occupa i caratteri dalla posizione 7 alla 12. Supponendo che il cursore non sia alla linea 5 quando il comando viene emesso, quale sequenza di escape ANSI deve essere emessa dall'editor per cancellare la parola?
35. Molti terminali RS-232 hanno delle sequenze di escape per la cancellazione della riga sulla quale si trova il cursore e per lo scorrimento in alto di una posizione di tutte le righe che si trovano al di sotto. Come potrebbe essere realizzata tale funzione all'interno del terminale?
36. Sul video a colori del PC IBM originale la scrittura nella RAM video in un qualsiasi momento che non sia quello del ritorno verticale del pennello elettronico causa l'apparizione di fastidiosi puntini su tutto lo schermo. La schermata è composta da 25 caratteri per 80, ognuno dei quali sta all'interno di un'area di 8 per 8 pixel. Ogni riga di 640 pixel viene tracciata in una singola scansione orizzontale del video da parte del pennello elettronico, la quale viene effettuata in 63,6 μ s, compreso il ritorno orizzontale. Lo schermo viene tracciato 60 volte al secondo, e ogni volta ci vuole un periodo di tempo per riportare il pennello in cima allo schermo. Per quale frazione di tempo la RAM video risulta disponibile per la scrittura?

37. I progettisti di un sistema si aspettavano che il mouse venisse mosso alla velocità massima di 20 cm/s. Se un mickey misura 0,1 mm e ogni messaggio di mouse è di 3 byte, qual è la frequenza di dati massima del mouse, supponendo che ogni mickey venga segnalato separatamente?
38. I colori additivi primari sono il rosso, il verde e il blu, il che significa che ogni colore può essere costruito sovrapponendo linearmente questi colori. È possibile produrre una fotografia a colori che non possa essere rappresentata usando colori a 24 bit?
39. Un modo di inserire un carattere in uno schermo a mappa di bit è usare bitblt da una tabella di font. Se un particolare font usa caratteri formati da 16x24 pixel in true RGB.
- Quanto spazio della tabella di font viene occupato da ogni carattere?
 - Se copiare un byte richiede 100 ns, incluso l'overhead, qual è la velocità di output sullo schermo in caratteri al secondo?
40. Sapendo che copiare un byte richiede 10 ns, quanto tempo occorre per riscrivere uno schermo di 80 caratteri per 25 linee in modo testo su uno schermo mappato in memoria? Cosa succede nel caso di uno schermo grafico di 1024x768 pixel con colori a 24 bit?
41. In Figura 5.41 c'è una classe *Register Class* (classe registro). Nel corrispondente codice X Windows, in Figura 5.46, non compare nulla di simile. Perché?
42. Nel testo abbiamo dato un esempio di come si possa disegnare un rettangolo su uno schermo usando la GDI di Windows:
- ```
Rectangle(hdc, xsin, yalto, xdestra, ybasso);
```
- C'è un vero bisogno del primo parametro (*hdc*) e, se sì, qual è tale bisogno? Dopo tutto, le coordinate del rettangolo sono esplicitamente passate come parametri.
43. Un terminale SLIM viene usato per visualizzare una pagina Web contenente un cartone animato di dimensione 400 pixel x 160 pixel a 10 frame al secondo. Quale frazione di una Fast Ethernet a 100 Mbps viene consumata nella visualizzazione del cartone animato?
44. È stato osservato che il sistema SLIM lavora bene con una rete di 1 Mbps. Ci sono problemi possibili, nel caso multiutente? Suggerimento: considerare un grande numero di utenti che guardano un programma televisivo e lo stesso numero di utenti che navigano su Internet.
45. Se il voltaggio massimo di una CPU,  $V$ , viene tagliato a  $V/n$ , il suo consumo scende a  $1/n^2$  del suo valore originale e la sua velocità di clock a  $1/n$  del suo valore originale. Supponi che un utente stia scrivendo a 1 carattere al secondo, ma il tempo di CPU richiesto per elaborare ogni carattere è di 100 ms. Qual è il valore ottimale di  $n$  e qual è il corrispondente risparmio di energia in percentuale, rispetto a non ridurre il voltaggio? Supponi che una CPU inattiva non consumi energia.
46. Un computer portatile è impostato in modo da sfruttare al massimo le caratteristiche di risparmio energetico, come lo spegnimento del display e del disco fisso dopo un certo periodo di inattività. Un utente talvolta usa programmi UNIX in modo testo e altre volte il sistema X Window; è sorpresa del fatto che la vita della batteria aumenta in modo significativo quando usa programmi di solo testo. Perché?
47. Scrivi un programma che simuli la memorizzazione stabile. Usa due grandi file di lunghezza fissata sul tuo disco per emulare i due dischi.



# I FILE SYSTEM

Tutte le applicazioni su calcolatore hanno bisogno di memorizzare e rintracciare informazioni. Durante l'esecuzione, un processo può memorizzare una quantità limitata di informazioni nello spazio degli indirizzi ad esso concesso e, comunque, la capacità di memoria è limitata dalla dimensione dello spazio degli indirizzi virtuali. Per alcune applicazioni questo spazio riservato è adeguato, ma per altre, come prenotazioni aeree, applicazioni bancarie o archivi di grandi aziende, è spesso troppo limitato.

Un secondo problema nel mantenere le informazioni all'interno dello spazio degli indirizzi di un processo è che le informazioni vengono perse quando il processo termina; per molte applicazioni (ad esempio, per le basi di dati) le informazioni devono essere conservate per settimane, mesi o a volte per sempre, per cui è inaccettabile che esse possano essere perse quando il processo termina. Inoltre le informazioni non devono andare perse quando si guasta il computer e si interrompono i processi in corso in quell'istante.

Un terzo problema consiste nel fatto che frequentemente più processi devono poter accedere contemporaneamente ad una parte delle informazioni. Se abbiamo un elenco telefonico in linea memorizzato nello spazio degli indirizzi di un processo, solo quel processo può accedervi; il modo per risolvere questo problema consiste nel rendere le informazioni stesse indipendenti da ogni altro processo.

Abbiamo così individuato tre requisiti essenziali per la memorizzazione duratura delle informazioni:

1. Deve essere possibile memorizzare una grossa quantità di informazioni.
2. Le informazioni devono essere conservate anche dopo la terminazione del processo che le usa.

3. Più processi devono avere la possibilità di accedere alle informazioni in modo corrente.

La soluzione di tutti questi problemi consiste nel registrare le informazioni su dischi e su altri supporti esterni in unità dette **file**: se necessario, i processi possono leggere e scrivere nuovi file. Le informazioni memorizzate nei file devono essere **permanenti**, non devono cioè essere influenzate dalla creazione e dalla terminazione del processo; un file potrà essere distrutto solo quando il suo proprietario lo cancella esplicitamente.

I file sono gestiti dal sistema operativo; nella progettazione di un sistema operativo sono argomenti importanti la struttura, il nome, l'accesso, l'uso e la protezione dei file stessi. Quella parte del sistema operativo che si occupa complessivamente dei file è chiamata **file system** ed è l'argomento trattato in questo capitolo.

Dal punto di vista degli utenti, l'aspetto più importante di un file system è come si presenta ad essi, cioè cosa costituisce un file, come sono protetti, nominati, quali operazioni sono permesse sui file e così via. Alcuni particolari, quali l'uso di liste collegate (linked list) o mappe di bit (bit map) per tenere traccia della memoria libera, e il numero dei settori che fanno parte di un blocco logico, sono meno interessanti sebbene molto importanti per la progettazione del file system. Per questa ragione il capitolo è stato suddiviso in molti paragrafi: i primi due paragrafi riguardano rispettivamente l'interfaccia utente dei file e delle directory. Segue una discussione dettagliata sull'implementazione del file system, dopo di che si faranno alcuni esempi di file system reali.

## 6.1 I file

In questo paragrafo si studieranno i file dal punto di vista dell'utente, cioè il loro uso e le loro proprietà.

### 6.1.1 Denominazione dei file

I file sono un meccanismo di astrazione: essi offrono un modo di memorizzare le informazioni su disco e di ritrovarle in un secondo momento, in modo che l'utente non debba interessarsi a come e a dove le informazioni sono memorizzate, ed a come funzionano realmente i dischi.

Probabilmente la caratteristica più importante di ogni meccanismo di astrazione è il modo in cui gli oggetti gestiti sono denominati, per cui si inizierà ad esaminare il file system dal problema della denominazione dei file. Quando un processo crea un file, esso dà un nome al file; quando il processo termina, il file continua ad esistere, ed altri processi possono utilizzare le sue informazioni usando il nome.

Le regole precise per denominare i file possono variare da sistema a sistema, ad ogni modo tutti i sistemi operativi odierni permettono l'uso di stringhe lunghe da uno a otto lettere come nomi legittimi dei file; pertanto *Andrea*, *Bruno* e *Carol* sono nomi possibili di file. Spesso sono anche permessi e ritenuti validi numeri e caratteri speciali: pertanto anche i nomi *2*, *urgenze!* e *Fig.2.14* spesso sono ammessi. Molti file system consentono nomi lunghi fino a 255 caratteri.

Alcuni file system distinguono tra lettere maiuscole e minuscole, altri no: UNIX appartiene alla prima categoria; MS-DOS appartiene alla seconda categoria. Così succede che un sistema UNIX può avere tre file distinti di nome: *maria*, *Maria* e *MARIA*, mentre in MS-DOS tutti i nomi precedenti rappresentano lo stesso file.

Una digressione sui file system è probabilmente doverosa, a questo punto. Windows 95 e Windows 98 usano il file system di MS-DOS e quindi hanno ereditato molte delle sue proprietà, tra le quali il modo in cui i nomi dei file vengono costruiti; inoltre, Windows NT e Windows 2000 supportano il file system di MS-DOS e quindi anch'essi ereditano alcune sue proprietà. Però, gli ultimi due hanno un file system nativo (NTFS) che ha diverse proprietà (ad esempio i nomi dei file in Unicode). In questo capitolo, quando ci riferiamo ai file system di Windows, intendiamo il file system di MS-DOS, che è l'unico file system supportato da tutte le versioni di Windows, mentre discuteremo il file system nativo di Windows 2000 nel Capitolo 11.

Molti sistemi operativi dividono il nome del file in due parti, con la seconda parte separata da un punto, come in *prog.c*. La parte che segue il punto si chiama estensione del file (file extension) e di solito indica qualche informazione relativa al file. In MS-DOS, per esempio, i nomi dei file sono composti da 1 a 8 caratteri, più una estensione opzionale da 1 a 3 caratteri. In UNIX, la dimensione dell'estensione, se presente, è lasciata all'utente e un file può avere due o più estensioni, come in *prog.c.Z* dove *.Z* è usata di solito per indicare che il file (*prog.c*) è stato compresso usando l'algoritmo di compressione di Ziv-Lempel. Alcune estensioni di file tipiche e il loro significato sono mostrati in Figura 6.1.

In alcuni sistemi (ad esempio, UNIX) l'estensione del file è solo una convenzione e non è fatta valere in nessun modo dal sistema. Un file denominato *file.txt* è probabilmente un qualche tipo di file di testo (text), ma questo nome serve da promemoria all'utente più che per trasmettere qualche informazione specifica al computer. D'altra parte un compilatore C può pretendere che i file da compilare abbiano l'estensione *.c* e in caso contrario rifiutarsi di effettuare la compilazione.

Questo tipo di convenzioni sono utili specialmente quando lo stesso programma può manipolare diversi tipi di file. Il compilatore C, per esempio, può ricevere in ingresso una lista di file da compilare e collegare insieme, di cui alcuni sono file C, e altri sono file in linguaggio assembler. A questo punto l'estensione diventa essenziale per il compilatore per sapere quali file sono C, quali assembler, e quali sono altri tipi di file.

Al contrario, Windows fa caso alle estensioni e dà loro significato. Gli utenti (o i processi) possono registrare estensioni nel sistema operativo, e specificare per ciascuna qua-

| Estensione       | Significato                                             |
|------------------|---------------------------------------------------------|
| <i>file.bak</i>  | File di backup                                          |
| <i>file.c</i>    | Programma sorgente C                                    |
| <i>file.gif</i>  | Formato per immagini GIF (Graphical Interchange Format) |
| <i>file.hlp</i>  | File di aiuto                                           |
| <i>file.html</i> | Documento per World Wide Web HyperText Markup Language  |
| <i>file.jpg</i>  | Foto codificata con lo standard JPEG                    |
| <i>file.mp3</i>  | Musica codificata in MPEG livello 3, formato audio      |
| <i>file.mpg</i>  | Video codificato con lo standard MPEG                   |
| <i>file.o</i>    | File oggetto [compilato, ma non ancora collegato]       |
| <i>file.pdf</i>  | Documento formato portatile (Portable Document Format)  |
| <i>file.ps</i>   | File PostScript                                         |
| <i>file.tex</i>  | Input per il programma di formattazione TEX             |
| <i>file.txt</i>  | File di testo generico                                  |
| <i>file.zip</i>  | Archivio compresso                                      |

Figura 6.1 Alcune estensioni di file tipiche.

le programma "possiede" i file terminanti in tale estensione; quando l'utente dà un doppio click su un nome di file, il programma assegnato a quella particolare estensione viene lanciato con il file stesso come parametro. Ad esempio un doppio click su *file.doc* fa partire *Word* di Microsoft con *file.doc* come file aperto inizialmente.

### 6.1.2 Struttura dei file

I file possono essere strutturati in molti modi. Tre possibilità comunemente usate sono mostrate in Figura 6.2. Il file di Figura 6.2(a) è una sequenza non strutturata di byte; in effetti al sistema operativo non interessa sapere cosa contiene il file e tutto ciò che vede sono byte. Un qualsiasi significato deve essere attribuito a livello di programma utente. Sia UNIX sia Windows utilizzano questo tipo di approccio.

Se il sistema operativo tratta i file come se non fossero nient'altro che una sequenza di byte, si ottiene la massima flessibilità da parte sua. I programmi utente possono inserire nei file qualsiasi cosa e dar loro un nome secondo quanto ritengono opportuno. Il sistema operativo non fornisce nessun aiuto, ma non interferisce nemmeno; per gli utenti che vogliono fare cose insolite ciò può essere molto importante.

La prima evoluzione della struttura di un file è mostrata in Figura 6.2(b). In questo modello, un file è una sequenza di record a lunghezza fissa, ognuno con una sua struttura interna. L'idea fondamentale di un file come sequenza di record è che l'operazione di lettura restituisca un record e che le operazioni di scrittura riscrivano o aggiungano un record. Come nota storica si può dire che, negli anni passati, quando imperavano le schede perforate a 80 colonne, molti sistemi operativi (per mainframe) basavano il loro file system su file composti da record di 80 caratteri, a immagine delle schede. Questi sistemi trattavano anche file composti da record di 132 caratteri, che erano usati per una riga di stampa (che a quel tempo aveva 132 colonne). I programmi leggevano l'input su blocchi di 80 caratteri e lo scrivevano su blocchi di 132 caratteri, sebbene naturalmente gli ultimi 52 avrebbero potuto essere spazi. Nessun sistema di uso generale odierno lavora in questo modo.

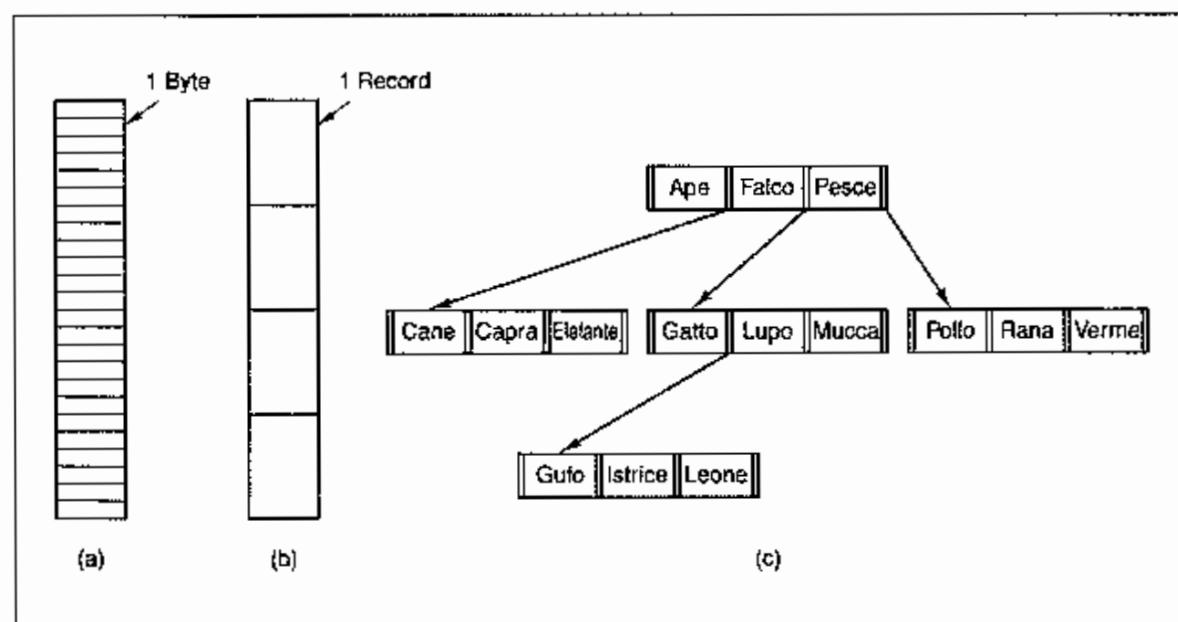


Figura 6.2. Tre tipi di file. (a) Sequenza di byte. (b) Sequenza di record. (c) Albero.

Il terzo tipo di struttura di file è mostrato in Figura 6.2(c). In questa organizzazione, un file è formato da un albero di record, non tutti necessariamente della stessa lunghezza, ognuno contenente un campo **chiave** (key field) in una posizione ben definita nel record. L'albero è ordinato rispetto al campo chiave, permettendo una ricerca veloce per una chiave determinata.

In questo caso l'operazione di base non è accedere al record "successivo", sebbene ciò sia anche possibile, bensì accedere al record individuato da una chiave specifica. Per il file *zoo* di Figura 6.2(c), si potrebbe chiedere al sistema l'accesso al record la cui chiave è *pollo*, per esempio, senza preoccuparsi della sua posizione esatta all'interno del file; inoltre si possono aggiungere al file nuovi record utilizzando il sistema operativo senza che l'utente debba decidere dove disporli. Questo tipo di file chiaramente è molto diverso dai flussi di byte non strutturati usati in UNIX e in Windows, ed è impiegato largamente sui grossi mainframe, usati ancora oggi per l'elaborazione dati commerciale.

### 6.1.3 Tipi di file

Diversi sistemi operativi trattano molti tipi di file. UNIX e Windows, per esempio, hanno file regolari e directory. UNIX ha anche file di caratteri speciali e file di blocchi speciali. I file regolari sono quelli che contengono le informazioni dell'utente; tutti i file di Figura 6.2 sono file regolari. Le directory sono file di sistema che conservano la struttura del file system, e le analizzeremo più avanti. I file di caratteri speciali riguardano l'input/output e sono usati per modellare unità seriali di I/O come terminali, stampanti e reti. I file di blocchi speciali sono usati per modellare i dischi. In questo capitolo ci occuperemo principalmente dei file regolari.

Generalmente i file regolari sono o file di tipo ASCII oppure file di tipo binario. I file di tipo ASCII sono formati da linee di testo; in qualche sistema ogni linea termina con un carattere di invio (carriage return), in altri si usa il carattere line feed; occasionalmente (ad esempio in MS-DOS) si usano entrambi i caratteri e, ad ogni modo, non è necessario che le linee abbiano la stessa lunghezza.

Il grosso vantaggio dei file di tipo ASCII è che essi possono essere visualizzati e stampati così come sono e possono essere editati con un qualsiasi programma editore di testo; inoltre, se un gran numero di programmi usa file di tipo ASCII per l'input e l'output, è facile collegare l'output di un programma all'input di un altro, come nelle pipeline della shell (l'interconnessione tra processi non è per niente facile, ma se si usa una convenzione standard, come quella ASCII, allora interpretare le informazioni diventa più semplice).

Tutti gli altri file che non sono file ASCII sono file di tipo binario: stamparli sulla stampante produce un listato incomprensibile pieno di ciò che apparentemente sembra informazione casuale. Di solito questi file hanno una loro struttura interna, nota al programma che li usa.

Per esempio, in Figura 6.3(a) si vede un semplice file binario eseguibile preso da una versione UNIX. Sebbene tecnicamente il file sia una sequenza di byte, il sistema operativo lo eseguirà solo se esso ha il formato appropriato. Il file ha cinque sezioni: intestazione, testo, dati, bit di rilocazione e tabella dei simboli. L'intestazione inizia con una parte chiamata **magic number** (numero magico), che identifica il file come eseguibile (per impedire l'esecuzione accidentale di un file non in questo formato). Seguono alcuni interi che danno la dimensione delle varie parti del file, l'indirizzo da cui parte l'esecuzione e qualche flag. Dopo l'intestazione si trovano il testo e i dati del programma: queste parti sono caricate in memoria e rilocate usando i bit di rilocazione. La tabella dei simboli è usata per il debugging.

Il secondo esempio di file binario è un archivio, sempre di UNIX: è formato da un insieme di procedure di libreria (moduli), compilate ma non collegate. Ognuna è preceduta da una intestazione che ne descrive il nome, la data di creazione, il proprietario, il codice di protezione e la dimensione. Come per i file eseguibili, le intestazioni dei moduli contengono numeri binari: inviandoli su una stampante si ottiene una stampa assolutamente priva di senso.

Tutti i sistemi operativi devono riconoscere un tipo di file, il loro file eseguibile, ma qualche volta ne riconoscono altri. Il vecchio sistema TOP-20 (per il DECsystem 20) arrivava ad esaminare il tempo di creazione di ogni file da eseguire, per poi individuare il file sorgente e guardare se era stato modificato dopo che era stato creato il binario; in questo caso avrebbe automaticamente ricompilato il sorgente. In termini UNIX, il programma *make* era stato ricostruito nella shell. Le estensioni dei file erano obbligatorie perché il sistema operativo potesse sapere da quale file sorgente era stato derivato un certo file binario.

Questa forte tipizzazione dei file provoca problemi ogni volta che l'utente fa qualcosa che i progettisti del sistema non avevano previsto. Come esempio si consideri un sistema in cui i file di output del programma hanno come tipo *.dat* (file dati); se un utente scrive un programma di formattazione che legge un file *.c* (un programma C), lo trasforma (per esempio usando un formato standard di indentazione), e poi scrive in output il file tra-

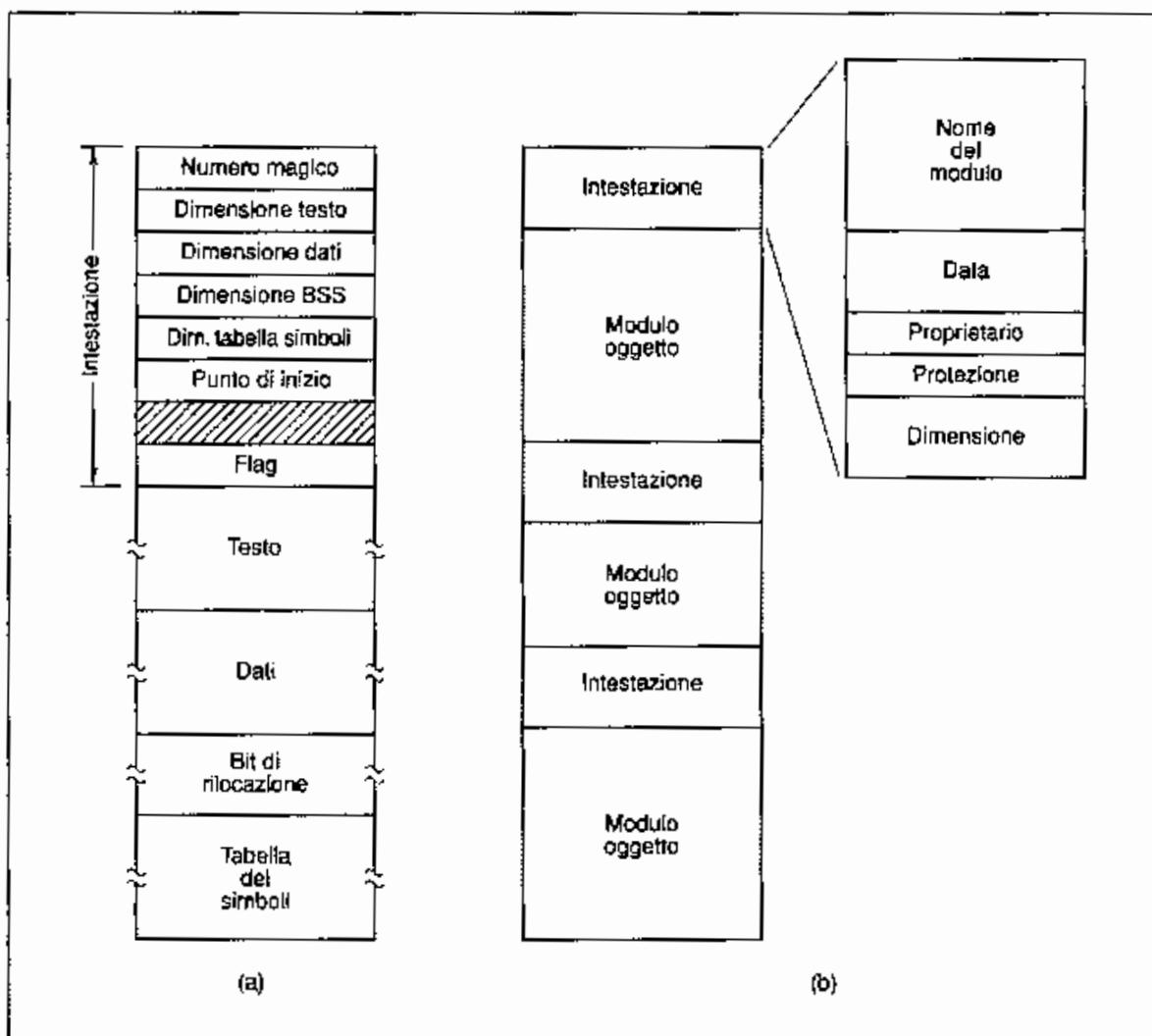


Figura 6.3. (a) Un file eseguibile. (b) Un archivio.

sformato, tale file sarà di tipo *.dat*; se l'utente dà questo programma a un compilatore C, il sistema lo rifiuterà perché ha un'estensione sbagliata. Tentare di copiare *file.dat* in *file.c* è un'operazione che il sistema si rifiuta di effettuare in quanto la considera un'operazione non valida (per proteggere nuovamente l'utente da errori).

Mentre questo tipo di "facilità d'uso" può aiutare i principianti, costringe gli utenti esperti ad arrampicarsi sugli specchi, dedicando sforzi considerevoli per aggirare la nozione del sistema operativo riguardo a cosa è lecito e cosa non lo è.

## 6.1.4 Accesso ai file

I primi sistemi operativi offrivano solo un tipo di accesso ai file: l'**accesso sequenziale**. In questi sistemi, un processo poteva leggere tutti i byte o i record di un file in ordine, partendo dall'inizio, ma non era possibile saltarne qualcuno o leggerli fuori dall'ordine. I file sequenziali possono essere riavvolti, comunque, in modo da poterli rileggere tutte le volte che bisogna farlo; sono convenienti quando l'unità di memoria è il nastro magnetico, piuttosto che il disco.

Quando si cominciò a usare i dischi per memorizzare i file, diventò possibile leggere i byte o i record di un file in modo non ordinato, o accedere ai record con la chiave piuttosto che con la loro posizione. I file i cui byte o record possono essere letti in qualunque ordine sono detti **file ad accesso casuale**; essi sono richiesti in molte applicazioni.

I file ad accesso casuale sono indispensabili per molte applicazioni, ad esempio nei sistemi di basi di dati. Se un cliente di una linea aerea chiama e vuole prenotare un posto su un volo particolare, il programma di prenotazioni deve essere in grado di accedere al record di quel determinato volo senza dover prima leggere i record di migliaia di altri voli.

Per specificare da dove partire con la lettura si usano due metodi. Nel primo, ogni operazione di lettura dà la posizione nel file da cui partire con la lettura; nel secondo, una operazione speciale, **seek**, dà la possibilità di selezionare la posizione corrente. Dopo aver fatto ciò, il file può essere letto sequenzialmente dalla nuova posizione corrente.

In qualche sistema operativo dei mainframe più vecchi, i file sono classificati ad accesso sequenziale o casuale al momento della loro creazione e ciò permette al sistema operativo di utilizzare tecniche di memorizzazione differenti per le due classi. I sistemi operativi moderni non fanno questa distinzione, tutti i file da loro trattati sono automaticamente ad accesso casuale.

## 6.1.5 Attributi dei file

Ogni file possiede un nome e i suoi dati; inoltre, tutti i sistemi operativi associano altre informazioni ad ogni file, per esempio, la data e l'ora della creazione e la sua dimensione; queste informazioni extra sono chiamate **attributi** del file. La lista degli attributi varia considerevolmente da sistema a sistema; la tabella di Figura 6.4 mostra alcune possibilità, anche se ne esistono altre. Nessun sistema esistente le comprende tutte, ma ognuna è presente in qualche sistema.

I primi quattro attributi riguardano la protezione dei file e indicano chi può accedervi e chi no. Sono possibili tutti i tipi di schemi, alcuni dei quali saranno studiati più avanti. In qualche sistema l'utente deve usare una password per accedere a un file, nel qual caso essa deve essere un attributo.

I flag sono bit o campi corti che controllano o abilitano qualche proprietà specifica; i file nascosti, per esempio, non compaiono nella lista di tutti i file. Il flag di archiviazione è un bit che mantiene traccia del fatto che il file è stato salvato; il programma di ripristino lo azzera, il sistema operativo lo mette a 1 ogni volta che un file è modificato, e in que-

| Attributo                | Significato                                                 |
|--------------------------|-------------------------------------------------------------|
| Protezione               | Chi può accedere al file e in che modo                      |
| Password                 | Parola chiave necessaria per accedere al file               |
| Creatore                 | ID della persona che ha creato il file                      |
| Proprietario             | Proprietario corrente                                       |
| Flag sola lettura        | 0 per lettura/scrittura; 1 per sola lettura                 |
| Flag file nascosto       | 0 per normale; 1 per non mostrare nelle liste               |
| Flag di sistema          | 0 per file normali; 1 per file di sistema                   |
| Flag di archivio         | 0 se il file ha subito backup; 1 altrimenti                 |
| Flag ASCII/binario       | 0 per file ASCII; 1 per file binario                        |
| Flag accesso casuale     | 0 per solo accesso sequenziale; 1 per accesso casuale       |
| Flag temporaneo          | 0 per normale; 1 cancella il file all'uscita del processo   |
| Flag di bloccaggio       | 0 per sbloccato; diverso da zero per bloccato               |
| Lunghezza record         | Numero di byte in un record                                 |
| Posizione chiave         | Offset della chiave all'interno di ogni record              |
| Lunghezza chiave         | Numero di byte nel campo chiave                             |
| Tempo creazione          | Data e ora in cui il file fu creato                         |
| Tempo ultimo accesso     | Data e ora dell'ultimo accesso al file                      |
| Tempo ultimo cambiamento | Data e ora in cui il file è stato modificato l'ultima volta |
| Dimensione corrente      | Numero di byte del file                                     |
| Dimensione massima       | Numero di byte fino al quale il file può crescere           |

Figura 6.4 Alcuni attributi di file possibili.

sto modo il programma di ripristino può sapere quali file hanno bisogno di essere salvati. Il flag temporaneo permette di marcare il file per la cancellazione automatica, quando termina il processo che lo aveva creato.

La lunghezza del record, la posizione della chiave e la lunghezza della chiave sono campi presenti solo nei file i cui record possono essere ritrovati usando una chiave: essi forniscono l'informazione necessaria per trovare le chiavi.

Le varie indicazioni di orario mantengono una traccia del momento in cui il file è stato creato, più recentemente visitato e modificato. Tutto ciò è utile per vari scopi; per esempio un file sorgente che è stato modificato dopo la creazione del file oggetto corrispondente dovrà essere ricompilato. Questi campi forniscono le informazioni necessarie.

La dimensione corrente indica la grandezza del file in quel momento. I vecchi sistemi operativi di qualche mainframe richiedevano di specificare la dimensione massima quando si creava il file, per permettere al sistema operativo di riservare in anticipo tale quantità massima di memoria; i sistemi di mini e personal computer sono abbastanza evoluti da gestire i file senza bisogno di queste informazioni.

## 6.1.6 Operazioni sui file

I file servono a memorizzare informazioni e a permettere che esse siano rintracciate in un secondo momento; diversi sistemi consentono operazioni diverse per memorizzarle e ricuperarle. Segue una discussione delle chiamate di sistema più usate relative ai file.

1. **Create.** Il file è creato senza dati. Lo scopo di questa chiamata è informare che il file sta per essere creato e selezionare alcuni attributi.

2. **Delete.** Quando il file non serve più, deve essere cancellato per liberare spazio su disco: per questo scopo c'è sempre una chiamata di sistema.
3. **Open.** Prima di usare un file, un processo lo deve aprire; lo scopo della chiamata open è permettere al sistema operativo di andare a prendere gli attributi e l'elenco degli indirizzi del disco nella memoria principale, per un accesso veloce nelle chiamate seguenti.
4. **Close.** Quando sono terminati tutti gli accessi, gli attributi e gli indirizzi del disco non sono ulteriormente necessari, così il file potrà essere chiuso per liberare spazio nella tabella interna; molti sistemi favoriscono ciò imponendo un numero massimo di file aperti sui processi. Un disco viene scritto in blocchi e la chiusura di un file forza la scrittura dell'ultimo blocco del file, anche se quel blocco può non essere completamente pieno.
5. **Read.** I dati sono letti dal file. Di solito i byte provengono dalla posizione corrente; il chiamante deve specificare la quantità di dati di cui ha bisogno, e fornire anche un buffer per contenerli.
6. **Write.** I dati vengono scritti nel file, di nuovo, normalmente alla posizione corrente. Se la posizione corrente corrisponde alla fine del file, si incrementa la sua dimensione; se la posizione corrente è nel mezzo del file, i dati esistenti vengono coperti dai nuovi e persi definitivamente.
7. **Append.** Questa chiamata è una forma restrittiva di write. Essa può solo aggiungere dati alla fine del file. I sistemi che forniscono un insieme minimo di chiamate di sistema generalmente non hanno l'append, ma molti sistemi forniscono molte maniere alternative di fare la stessa cosa, e qualche volta hanno l'append.
8. **Seek.** Per i file ad accesso casuale è necessario specificare un metodo per dire dove prendere i dati. Un approccio comune è una chiamata di sistema, seek, che riposiziona il puntatore alla posizione corrente in un posto specifico del file; completata questa chiamata, i dati si possono leggere o scrivere in quella posizione.
9. **Get attributes.** I processi spesso hanno bisogno di leggere gli attributi del file per fare il loro lavoro. Per esempio, il programma *make* di UNIX è usato comunemente per gestire lo sviluppo di progetti software che utilizzano parecchi file sorgenti; quando si invoca *make*, esso esamina il tempo di modifica di tutti i file sorgenti e oggetto, e pre-dispone un numero minimo di ricompilazioni necessarie per aggiornarli tutti. Per fare questo, deve guardare gli attributi e, in particolare, l'orario di modifica.
10. **Set attributes.** Qualche attributo è selezionabile dall'utente e può essere cambiato dopo che il file è stato creato: ciò è reso possibile da questa chiamata di sistema. Un esempio ovvio è l'informazione sulla modalità di protezione; generalmente molti flag rientrano in questa categoria.
11.  **Rename.** Spesso succede che un utente ha bisogno di cambiare il nome di un file esistente: questa chiamata di sistema lo rende possibile. Questa chiamata di sistema non sempre è strettamente necessaria, perché il file può sempre essere copiato in un nuovo file con altro nome, quindi il vecchio file può essere cancellato.

### 6.1.7 Un esempio di programma che usa chiamate di sistema sui file

In questa sezione esamineremo un semplice programma UNIX che copia un file dal suo sorgente a un file destinazione. Il testo del programma è in Figura 6.5. Il programma

ha funzionalità minime e un trattamento degli errori ancor più limitato, ma dà un'idea ragionevole del funzionamento di qualche chiamata di sistema relativa ai file.

Il programma, *copiafile*, può essere chiamato, per esempio, con il comando

```
copiafile abc xyz
```

per copiare il file *abc* sul file *xyz*. Se *xyz* è già esistente, sarà riscritto altrimenti sarà creato. Il programma deve essere chiamato esattamente con due argomenti, entrambi nomi legali di file.

Le quattro istruzioni *#include* all'inizio del programma causano l'inclusione nel programma di una gran quantità di definizioni e di intestazioni di funzione e sono necessarie per rendere il programma conforme agli standard internazionali. La linea successiva è l'intestazione della funzione *main*, richiesta da ANSI C, ma tutto ciò è irrilevante per i nostri scopi.

La prima istruzione *#define* è una definizione macro che definisce la stringa *DIM\_BUF* come una macro che si espande nel numero 4096. Il programma leggerà e scriverà in blocchi di 4096 byte. È considerata una buona pratica di programmazione dare nomi alle costanti come questa e usare i nomi invece delle costanti; questa convenzione non solo rende più facilmente leggibili i programmi ma ne rende più facile la manutenzione. La seconda *#define* determina chi può accedere al file di output.

Il programma principale è chiamato *main* e ha due argomenti, *argc* e *argv*; essi vengono messi a disposizione dal sistema operativo quando si chiama il programma. Il primo argomento indica quante parole sono presenti sulla linea di comando che ha chiamato il programma, e in questo programma vale 3. Il secondo è un vettore di puntatori agli argomenti. Nella chiamata scritta precedentemente, gli elementi di questo vettore conterranno puntatori ai seguenti valori:

```
argv[0] = "copiafile"
argv[1] = "abc"
argv[2] = "xyz"
```

È mediante questo vettore che il programma accede ai suoi argomenti.

Sono dichiarate cinque variabili: le prime due, *in\_fd* e *out\_fd*, conterranno i descrittori del file (file descriptor), interi restituiti quando il file è aperto. Le successive due, *cont\_lettura* e *cont\_scrittura*, sono le quantità di byte restituite dalle chiamate di sistema *read* e *write*, rispettivamente. L'ultima, *buffer*, è il buffer usato per contenere i dati letti e quelli che devono essere scritti.

La prima istruzione verifica se *argc* vale 3, in caso contrario termina con il codice di stato a 1; ogni codice di stato diverso da 0 indica che si è verificato un errore. Il codice di stato è il solo modo di segnalare gli errori in questo programma. Una versione professionale del programma normalmente stampa anche dei messaggi di errore.

Successivamente si prova ad aprire il file sorgente e a creare il file destinazione. Se l'apertura del file sorgente ha successo, il sistema assegna un intero a *in\_fd*, per identificare il file; le chiamate successive devono includere questo intero in modo che il sistema riconosca quale file utilizzare. Allo stesso modo, se il file destinazione è creato con successo, *out\_fd* prende il valore che lo identifica. Il secondo argomento della funzione *creat* seleziona la modalità di protezione; inoltre se la creazione o l'apertura falliscono, il descrittore del file corrispondente è posto a -1 e il programma termina con un codice di errore.

Ora comincia il ciclo di copia. Esso inizia provando a leggere 4KB di dati in *buffer* con una chiamata alla procedura di libreria *read*, che coinvolge la chiamata di sistema *read*. Il primo parametro identifica il file, il secondo dà il buffer e il terzo indica quanti byte

```

/* Programma di copia di un file. Controllo errori minimale. */

#include <sys/types.h> /* Includi i file header necessari */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]); /* Prototipo ANSI */

#define DIM_BUF 4096 /* Usa una dim. del buffer di 4096 byte */
#define OUTPUT_MODE 0700 /* Bit di protezione per file output */

int main(int argc, char *argv[])
{
 int in_fd, out_fd, cont_lettura, cont_scrittura;
 char buffer[DIM_BUF];

 if (argc != 3) exit(1); /* Errore sintattico se argc non è 3 */

 /* Apri il file di input e crea il file di output */
 in_fd = open(argv[1], O_RDONLY); /* Apri il file sorgente */
 if (in_fd < 0) exit(2); /* Se non può essere aperto, esci */
 out_fd = creat(argv[2], OUTPUT_MODE); /* Crea il file destinazione */
 if (out_fd < 0) exit(3); /* Se non può essere creato, esci */

 /* Ciclo di copia */
 while (TRUE) {
 cont_lettura = read(in_fd, buffer, DIM_BUF); /* Leggi un blocco di dati */
 if (cont_lettura <= 0) break; /* Se alla fine del file o errore, esci */
 cont_scrittura = write(out_fd, buffer, cont_lettura); /* Scrivi i dati */
 if (cont_scrittura <= 0) exit(4); /* cont_scrittura <= 0, è un errore */
 }

 /* Chiudi i file */
 close(in_fd);
 close(out_fd);
 if (cont_lettura == 0) /* Nessun errore all'ultima lettura */
 exit(0);
 else
 exit(5); /* Errore all'ultima lettura */
}

```

**Figura 6.5** Un semplice programma di copia di file.

devono essere letti; il valore assegnato a *cont\_lettura* dà il numero di byte effettivamente letti. Di solito questo valore sarà 4096, a meno che nel file non sia rimasta una quantità minore di byte; quando si raggiunge la fine del file il suo valore sarà 0. Se *cont\_lettura* assume un valore minore o uguale a zero, la copia non continua ed è eseguita l'istruzione *break* per terminare il ciclo (altrimenti senza fine).

La chiamata a *write* scrive il buffer sul file destinazione. Il primo parametro identifica il file, il secondo dà il buffer e il terzo indica quanti byte scrivere, come per la *read*. Si noti che il contatore di byte indica il numero di byte letti e non *DIM\_BUF*; tutto ciò è importante perché di solito l'ultima lettura non avrà letto 4096 byte, a meno che la lunghezza del file non sia esattamente un multiplo di 4KB.

Quando l'intero file è stato elaborato, la prima chiamata dopo la fine del file restituirà

0 nella variabile `cont_lettura`, causando la fine del ciclo; a questo punto i due file sono chiusi e il programma termina con uno stato che indica che è stato eseguito con successo.

Anche se le chiamate di Windows sono diverse da quelle di UNIX, la struttura generale di un programma a linea di comando Windows per effettuare la copia di un file è abbastanza simile a quella di Figura 6.5. Esamineremo le chiamate di Windows 2000 nel Capitolo 11.

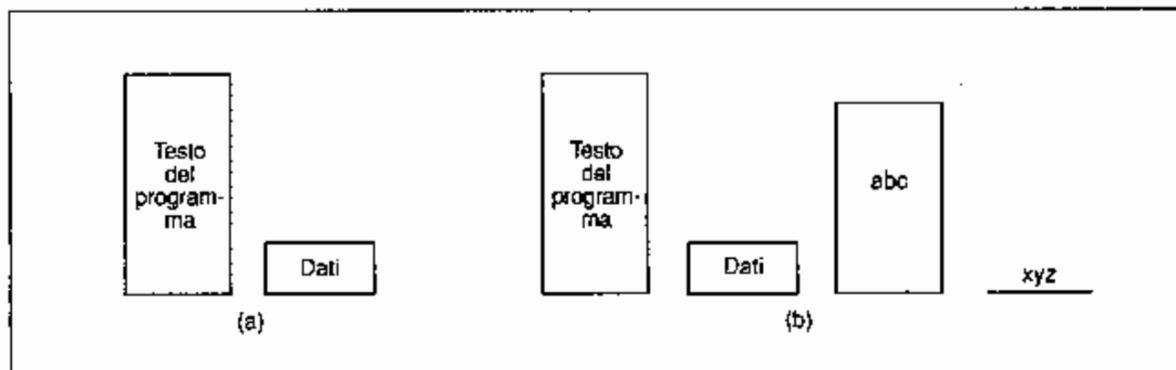
### 6.1.8 File mappati in memoria

È opinione comune che il modo di accedere ai file mostrato in precedenza sia scomodo e poco conveniente, specialmente quando paragonato con l'accesso alla memoria principale. Per questo motivo alcuni sistemi operativi, a partire da MULTICS, hanno messo a disposizione un modo per mappare i file nello spazio degli indirizzi di un processo in esecuzione. Concettualmente, si immagini l'esistenza di due nuove chiamate di sistema, `map` e `unmap`. La prima chiamata dà un nome di file e un indirizzo virtuale, in modo che il sistema operativo mappi il file nello spazio degli indirizzi all'indirizzo virtuale.

Per esempio, supponiamo che un file `f`, lungo 64K, sia mappato nell'indirizzo virtuale 512K: ogni istruzione macchina che legge il contenuto del byte all'indirizzo 512K prende il byte 0 del file e così via. Allo stesso modo una scrittura all'indirizzo 512K + 21000 modifica il byte 21000 del file. Quando il processo termina, il file viene posto sul disco, come se fosse stato modificato da una combinazione di chiamate di sistema `seek` e `write`.

Quello che realmente succede è che le tabelle interne del sistema vengono cambiate per far sì che il file serva come memoria di salvataggio per la regione di memoria che va da 512K a 576K. Perciò una lettura dall'indirizzo 512K causa un fault di pagina, che fa caricare la pagina 0 del file; allo stesso modo una scrittura all'indirizzo 512K + 1100 causa un fault di pagina, che fa caricare la pagina che contiene quell'indirizzo, dopodiché può avvenire la scrittura in memoria. Se quella pagina non viene mai tolta dalla memoria dall'algoritmo di sostituzione delle pagine, essa è riscritta nel posto appropriato del file, e quando il processo termina, tutte le pagine modificate e inappate sono riscritte nei loro file.

La mappatura dei file si realizza meglio in un sistema che è fornito di segmentazione. In un tale sistema ogni file può essere mappato in un suo segmento, cosicché il byte  $k$  nel file è anche il byte  $k$  nel segmento. La Figura 6.6(a) mostra un processo che ha due segmenti, `testo del programma` e `Dati`. Supponiamo che questo processo copi i file, come il programma di Figura 6.5: prima esso mappa il file sorgente, diciamo `abc`, in un segmento; poi crea un



**Figura 6.6.** (a) Un processo segmentato prima che i file vengano mappati nel suo spazio di indirizzamento. (b) Il processo dopo che un file esistente `abc` è stato mappato in un segmento, e dopo che è stato creato un nuovo segmento per il file `xyz`.

segmento vuoto e lo mappa nel file destinazione, nel nostro esempio *xyz*. Queste operazioni danno la situazione mostrata in Figura 6.6(b).

A questo punto il processo può copiare il segmento sorgente nel segmento destinazione usando un normale ciclo di copia. Per far ciò non occorrono chiamate di sistema *read* o *write*. Alla fine, il sistema può eseguire una chiamata *unmap* per rimuovere i file dallo spazio degli indirizzi e quindi terminare: il file di output, *xyz*, ora esisterà come se fosse stato creato convenzionalmente.

Sebbene la mappatura dei file elimini il bisogno di I/O e perciò renda più facile la programmazione, essa introduce alcuni problemi. Per prima cosa, il sistema ha difficoltà a conoscere la lunghezza esatta del file di output, nel nostro esempio *xyz*: il sistema può dire facilmente qual è il numero della pagina con indirizzo maggiore scritta finora, ma non ha alcuna possibilità di sapere quanti byte sono stati scritti in quella pagina. Si supponga che il programma usi solo la pagina 0 e che dopo l'esecuzione tutti i byte siano ancora 0 (il loro valore iniziale); forse *xyz* è un file formato da 10 o da 100 zeri, o magari da 1000 zeri. Chi può saperlo? Senz'altro non il sistema operativo; tutto ciò che può fare è creare un file la cui lunghezza è uguale alla dimensione della pagina.

Un secondo problema potenziale si può presentare se un file è mappato in memoria da un processo, e aperto per una lettura convenzionale da un altro. Se il primo processo modifica una pagina quella modifica non sarà riportata nel file su disco finché la pagina non è tolta dalla memoria; il sistema deve accuratamente evitare che i due processi usino versioni inconsistenti del file.

Un terzo problema è che un file può essere più grande di un segmento o anche dell'intero spazio degli indirizzi virtuali. Il solo modo per risolvere questo problema è invocare la chiamata di funzione *map* affinché mappi una porzione di un file, piuttosto che il file intero; anche se corretta, questa soluzione è chiaramente meno soddisfacente della mappatura del file stesso.

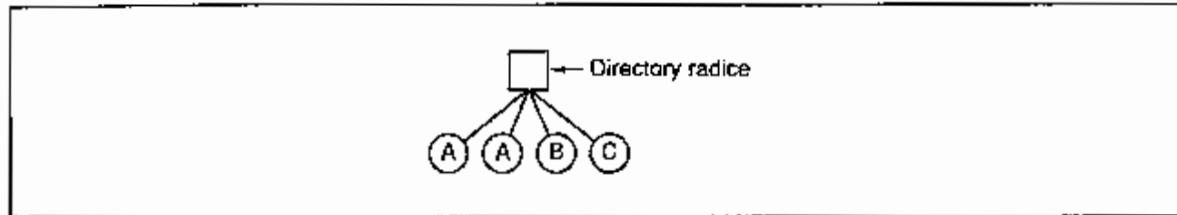
## 6.2 Le directory

Per tener traccia dei file, il file system normalmente fornisce le directory o cartelle che, in molti sistemi, sono a loro volta dei file. In questa sezione parleremo delle directory, della loro organizzazione, delle loro proprietà e delle operazioni che si possono effettuare su di esse.

### 6.2.1 Sistemi di directory a singolo livello

La forma più semplice di sistema di directory è quello di avere un'unica directory che contiene tutti i file; talvolta è chiamata directory radice, ma siccome è l'unica, il nome non è molto importante. Sui primi personal computer, questo sistema era comune, in parte perché c'era un unico utente. Stranamente, anche il primo supercomputer del mondo, il CDC 6600, aveva un'unica directory per tutti i file, anche se veniva usato da più di un utente per volta. Questa decisione era stata sicuramente presa per mantenere semplice il progetto del software.

Un esempio di sistema con una directory è mostrato in Figura 6.7; in questo esempio la directory contiene quattro file. Nella figura sono mostrati i proprietari e non i nomi dei file (in quanto i proprietari sono importanti in quello che stiamo per dire). I vantaggi di questo schema sono la sua semplicità e l'abilità di localizzare i file velocemente – c'è un unico posto dove cercare, dopo tutto.



**Figura 6.7** Un sistema di directory a singolo livello, contenente quattro file, di proprietà di tre persone diverse: A, B e C.

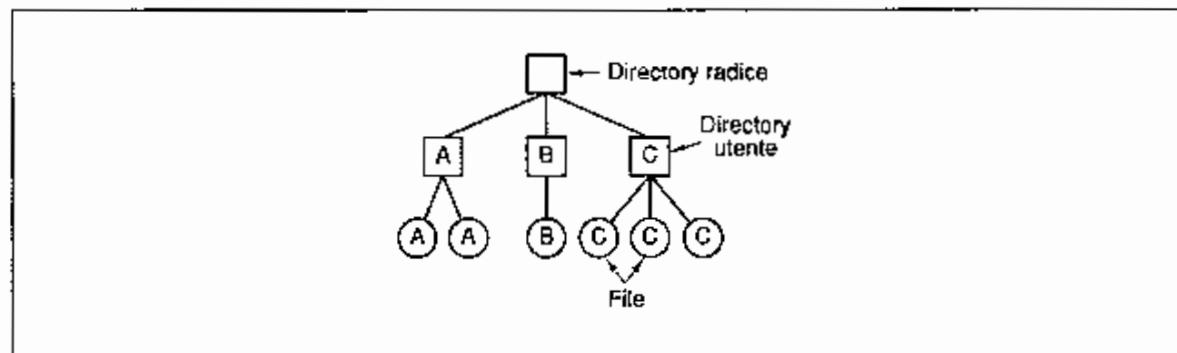
Il problema nell'avere un'unica directory in un sistema con più di un utente è che utenti diversi possono accidentalmente usare gli stessi nomi per i loro file: ad esempio se l'utente A crea un file chiamato *mailbox* ed, in seguito, anche l'utente B crea un file chiamato *mailbox*, il file dell'utente B sovrascriverà quello dell'utente A: di conseguenza questo schema non viene più usato per i sistemi multiutente, ma potrebbe essere ancora usato in un piccolo sistema dedicato, ad esempio un sistema in un'auto progettato per immagazzinare i profili degli utenti per un piccolo numero di guidatori.

## 6.2.2 Sistemi di directory a due livelli

Per evitare i conflitti causati da diversi utenti che scelgono gli stessi nomi di file per i loro file personali, il passo successivo è stato quello di dare ad ogni utente una directory privata. In questo modo, i nomi scelti da ogni utente non interferiscono coi nomi scelti da un altro, e non ci sono problemi se lo stesso nome è presente in due o più directory. Questa idea porta al sistema di Figura 6.8; questo schema di progettazione potrebbe essere usato, per esempio, su un sistema multiutente o su una semplice rete di PC che condividano un server di file comune, all'interno della stessa rete locale.

In questo schema è implicito che quando un utente cerca di aprire un file, il sistema sa di quale utente si tratti in modo da sapere in che directory cercare; di conseguenza, è necessaria una qualche procedura di ingresso, o *login*, nella quale l'utente specifica il nome della login o un'identificazione, cosa che non è richiesta in un sistema con directory a singolo livello.

Quando questo sistema viene implementato nella sua forma più semplice, gli utenti possono solo accedere ai file nella loro directory; una piccola estensione è quella di permettere agli utenti di accedere ai file di altri utenti, fornendo qualche informazione in più sul file che si desidera aprire. Ad esempio:



**Figura 6.8** Un sistema di directory a due livelli. Le lettere indicano i proprietari di file e directory.

```
open("x")
```

pôd essere una chiamata per aprire un file chiamato *x* nella directory dell'utente, mentre

```
open("nina/x")
```

pôd essere la chiamata che richiede l'apertura di un file *x* nella directory di un altro utente, Nina.

Una situazione in cui gli utenti hanno bisogno di accedere ai file di altri utenti è per eseguire programmi di sistema binari: avere copie delle stesse utility in ogni directory è chiaramente inefficiente. Quanto meno, c'è bisogno di una directory di sistema che contenga tutti i file binari eseguibili.

### 6.2.3 Sistemi di directory gerarchici

La gerarchia a due livelli elimina i conflitti sui nomi, ma non è soddisfacente per utenti con un grande numero di file; questo capita persino su un PC a singolo utente. È un'esigenza comune che un utente voglia raggruppare i propri file sulla base di una qualche logica: un professore, per esempio, può avere una collezione di file che insieme rappresentano un libro che sta scrivendo per un corso, una seconda collezione di file che contiene i programmi consegnati dagli studenti di un altro corso, un terzo gruppo di file contenente il codice di un sistema di compilazione avanzato che sta costruendo, un quarto gruppo che contiene proposte di finanziamento, ed altri file contenenti posta elettronica, resoconti di incontro, articoli che sta scrivendo, giochi e così via; è necessario un modo per raggruppare questi file in modo flessibile, secondo il desiderio dell'utente.

Ciò che è necessario è una gerarchia generale (ad esempio un albero di directory). Con questo approccio ogni utente può avere tante directory quante sono necessarie, così i suoi file possono essere raggruppati in modo naturale. Questo approccio è mostrato in Figura 6.9: qui le directory *A*, *B* e *C*, contenute nella directory radice, appartengono ognuna a un utente differente, due dei quali hanno creato sottodirectory per progetti su cui stanno lavorando.

Dare agli utenti la possibilità di creare un numero arbitrario di sottodirectory, fornisce un mezzo di strutturazione molto potente per organizzare il loro lavoro; per questa ragione, quasi tutti i moderni file system sono organizzati in questo modo.

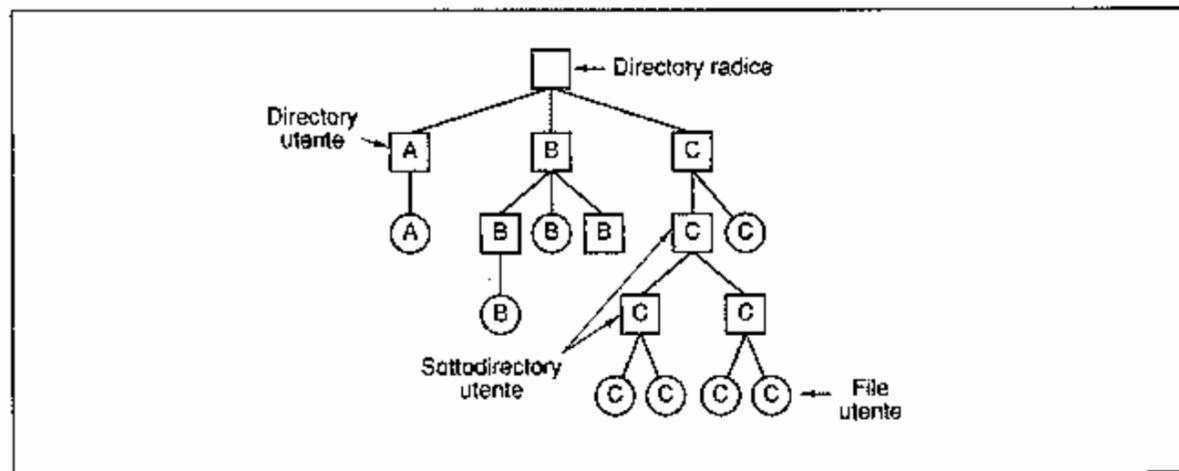


Figura 6.9 Un sistema gerarchico di directory.

## 6.2.4 I path name

Quando il file system è organizzato come un albero di directory, è necessario un qualche modo per specificare i nomi dei file; si usano di solito due metodi differenti. Nel primo metodo ad ogni file viene dato un **path name assoluto** (nome assoluto di cammino) che consiste nel cammino dalla directory radice al file. Come esempio, il cammino */usr/ast/mailbox* significa che la directory radice contiene una sottodirectory *usr*, che a sua volta contiene una sottodirectory *ast*, che contiene il file *mailbox*. I path name assoluti partono sempre dalla directory radice e sono unici. In UNIX i componenti del cammino sono separati da */*, in Windows il separatore è *\* e in MULTICS era *>*. Quindi lo stesso cammino potrebbe essere scritto, sui tre sistemi, nei seguenti tre modi diversi:

|         |                                  |
|---------|----------------------------------|
| Windows | <i>\usr\ast\mailbox</i>          |
| UNIX    | <i>/usr/ast/mailbox</i>          |
| MULTICS | <i>&gt;usr&gt;ast&gt;mailbox</i> |

Non importa quale carattere è usato: se il primo carattere del path name è il separatore, il cammino è assoluto.

L'altro tipo di nome è il **path name relativo**, che si usa congiuntamente al concetto di **directory di lavoro** (working directory, anche detta **directory corrente**). Un utente può definire una directory come la directory di lavoro corrente, nel qual caso tutti i path name che non iniziano dalla directory radice sono considerati relativi alla directory di lavoro. Per esempio, se la directory di lavoro corrente è */usr/ast*, allora il file il cui cammino assoluto è */usr/ast/mailbox* può essere indicato semplicemente con *mailbox*. In altre parole il comando UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

e il comando

```
cp mailbox mailbox.bak
```

fanno esattamente la stessa cosa se la directory di lavoro è */usr/ast*. La forma relativa è spesso più conveniente, ma ha lo stesso significato della forma assoluta.

Alcuni programmi hanno bisogno di accedere a un file specifico a prescindere dalla directory di lavoro; in quel caso, essi dovranno usare sempre path name assoluti. Per esempio, uno spelling checker (controllore lessicale) potrebbe aver bisogno di leggere */usr/lib/dizionario* per fare il suo lavoro; in questo caso dovrà usare il path name assoluto completo perché non sa quale sarà la directory di lavoro al momento della chiamata. Il path name assoluto andrà sempre bene, indipendentemente dalla directory di lavoro.

Naturalmente se lo spelling checker necessita di un gran numero di file da */usr/lib*, un approccio alternativo per esso è fare una chiamata di sistema per cambiare la propria directory di lavoro a */usr/lib*, quindi usare solo *dizionario* come primo parametro della chiamata di apertura del file. Cambiando esplicitamente la directory di lavoro, sa di sicuro dove è posizionato nell'albero delle directory, e può quindi usare i cammini relativi.

Ogni processo ha la sua propria directory di lavoro, così quando un processo cambia la sua directory di lavoro e più tardi termina, non ci sono altri processi coinvolti in questo cambiamento, e non ne rimane traccia nel file system. In questo modo è sempre perfettamente sicuro per un processo cambiare la sua directory di lavoro ogniqualvolta ciò è conveniente. D'altra parte se una *procedura di libreria* cambia la directory di lavoro e non ritorna dove era al momento della sua terminazione, il resto del programma non può fun-

zionare correttamente poiché le ipotesi fatte sulla sua posizione non sono più valide. Per questo motivo le procedure di libreria raramente cambiano la directory di lavoro, e quando devono farlo, la ripristinano sempre prima della loro terminazione.

La maggior parte dei sistemi operativi che adottano un sistema di directory gerarchico hanno due elementi speciali in ogni directory, “.” e “..”, generalmente chiamati “punto” e “punto punto”. Punto fa riferimento alla directory corrente; punto punto fa riferimento al suo genitore. Per vedere come questi vengono usati, si consideri l'albero di file UNIX in Figura 6.10. Un processo ha `/usr/ast` come directory di lavoro, quindi può usare .. per andare verso la radice dell'albero e, per esempio, può copiare il file `/usr/lib/dizionario` nella sua directory usando il comando

```
cp .. /lib/dizionario .
```

Il primo cammino dice al sistema di andare verso l'alto (verso la directory radice), poi di andare verso il basso nella directory `lib` per trovare il file `dizionario`.

Il secondo argomento (punto) indica la directory corrente. Quando il comando `cp` prende un nome di directory (compreso punto) come suo ultimo argomento, vi copia tutti i file. Naturalmente un modo più semplice di fare la copia potrebbe essere quello di scrivere:

```
cp /usr/lib/dizionario .
```

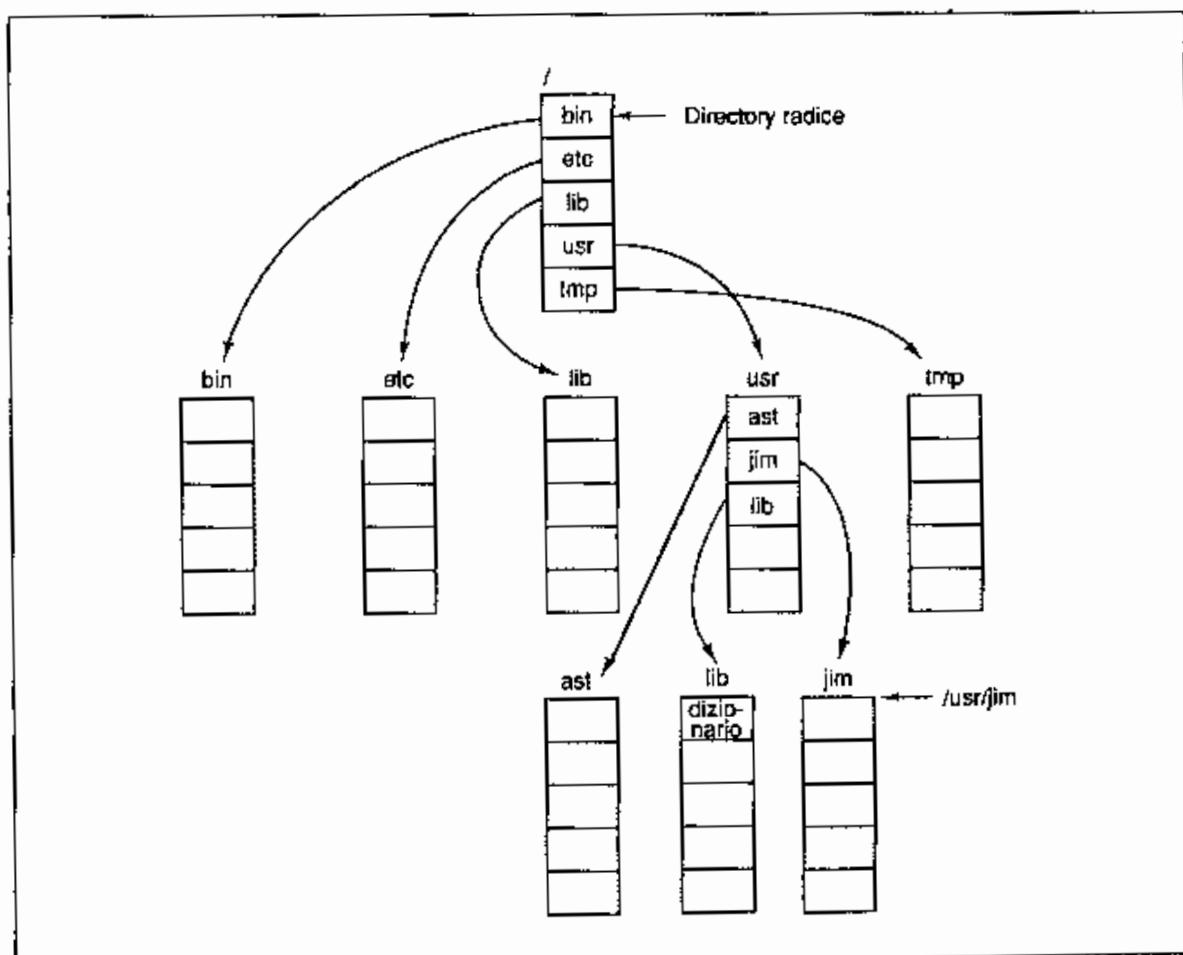


Figura 6.10 Una directory ad albero di UNIX.

Qui l'uso del punto evita all'utente di scrivere una seconda volta *dizionario*. Comunque, scrivere

```
cp /usr/lib/dizionario dizionario
```

funziona ugualmente, così come

```
cp /usr/lib/dizionario /usr/ast/dizionario
```

Tutti questi comandi fanno esattamente la stessa cosa.

## 6.2.5 Operazioni sulle directory

Le chiamate di sistema per la gestione delle directory cambiano maggiormente da sistema a sistema rispetto a quelle relative ai file. Ciononostante, per dare un'idea di quali sono e di come funzionano, si darà un esempio (preso da UNIX).

- Create.** Crea una directory vuota, a parte punto e punto punto che sono inseriti automaticamente dal sistema (o, in pochi casi, dal programma *mkdir*).
- Delete.** Cancella una directory. Solo una directory vuota può essere cancellata; una directory che contiene solo punto e punto punto è considerata vuota, perché questi elementi normalmente non possono essere cancellati.
- opendir.** Le directory si possono leggere; per esempio, per elencare tutti i file di una directory, un programma apre la directory per leggere i nomi di tutti i file che essa contiene. Prima che una directory possa essere letta, essa deve essere aperta, analogamente alla apertura e lettura dei file.
- Closedir.** Quando una directory è stata letta, dovrebbe essere chiusa per liberare lo spazio della tabella interna.
- Readdir.** Questa chiamata restituisce il prossimo elemento in una directory aperta. In passato era possibile leggere directory usando la chiamata di sistema *read*, ma tale approccio ha lo svantaggio di forzare il programmatore a conoscere e a occuparsi della struttura interna delle directory. Al contrario, *readdir* restituisce sempre un elemento in un formato standard, indipendentemente dalle possibili strutture di directory usate.
- Rename.** Sotto molti aspetti le directory sono come i file, e si possono ridenominare nello stesso modo dei file.
- Link.** Il collegamento è una tecnica che permette a un file di essere presente in più di una directory. Questa chiamata di sistema specifica un file esistente e un path name, e crea un link (collegamento) dal file esistente al nome specificato dal path: in questo modo, lo stesso file può essere presente in più directory. Un link di questo tipo, che incrementa il contatore nell'i-node del file (per tenere traccia del numero di directory che contengono il file), è talvolta chiamato **hard link** (link fisico).
- Unlink.** Un elemento di directory è rimosso. Se il file che è stato sganciato è presente solo in una directory (il caso normale), esso è rimosso dal file system. Se è presente in più directory, viene rimosso solo il path name specificato, gli altri restano. In UNIX, la chiamata di sistema per cancellare i file (discussa precedentemente) è infatti *unlink*.

La lista appena citata contiene le chiamate più importanti, ma ve ne sono altre, per esempio, per la gestione delle informazioni di protezione associate a una directory.

## 6.3 Implementazione del file system

Ora è giunto il momento di passare dal punto di vista dell'utente del file system al punto di vista di chi lo realizza (l'implementatore). Gli utenti si preoccupano di come i file sono chiamati, quali operazioni sono permesse su di essi, come si visita l'albero di directory e altri aspetti dell'interfaccia. Gli implementatori sono interessati al modo in cui i file e le directory sono memorizzate, alla gestione dello spazio su disco e al modo in cui far funzionare ogni cosa in modo efficiente e affidabile. Nelle prossime sezioni si esamineranno questi argomenti per studiarne gli aspetti importanti e i compromessi necessari.

### 6.3.1 La struttura dei file system

I file system sono memorizzati in dischi; la maggior parte dei dischi possono essere divisi in una o più partizioni, con file system indipendenti per ogni partizione. Il settore 0 del disco è chiamato **MBR** (record di avvio, Master Boot Record) e viene usato per avviare il computer. La fine del MBR contiene la tabella delle partizioni, che contiene il punto di inizio e di fine di ogni partizione; una delle partizioni della tabella è marcata come attiva. Quando il computer è avviato, il BIOS legge ed esegue il programma MBR: la prima cosa che questo programma fa, è localizzare la partizione attiva, leggere il suo primo blocco, chiamato **blocco di avvio** (boot block) ed eseguirlo. Il programma nel blocco di avvio carica il sistema operativo contenuto nella partizione attiva; per uniformità, ogni partizione inizia con un blocco di avvio, anche se non contiene un sistema operativo avviabile, ed inoltre, potrebbe contenerne uno in futuro, quindi tenere pronto un blocco di avvio è comunque una buona idea.

A parte il fatto di iniziare tutti con un blocco di avvio, la struttura di una partizione varia enormemente da file system a file system. Spesso un file system conterrà alcuni degli elementi mostrati in Figura 6.11; il primo è il **superblocco**, che contiene tutti i parametri chiave relativi al file system, ed è copiato in memoria all'avvio del computer o appena si accede al file system. Le informazioni tipiche contenute nel superblocco includono un numero magico di identificazione del tipo di file system, il numero di blocchi del file system e altre informazioni amministrative essenziali.

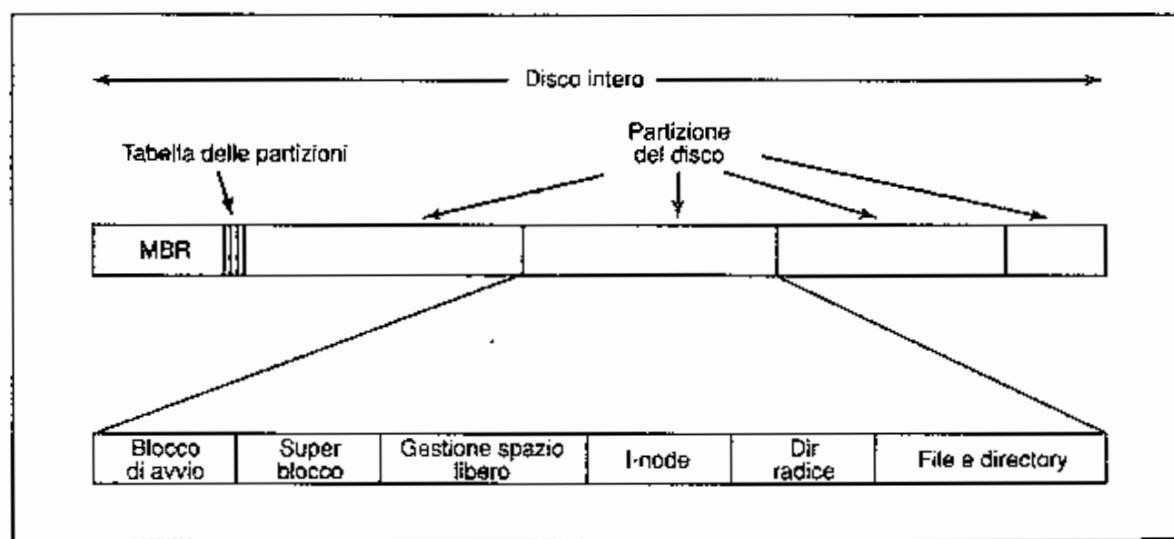


Figura 6.11 Una struttura possibile di file system.

In seguito può esserci qualche informazione circa i blocchi liberi, ad esempio una lista di puntatori o una bitmap, seguita dagli i-node, che sono un array di strutture dati, una per file, che contengono tutto quello che serve sapere su un file. Dopo di ciò può esservi la directory radice, che contiene la cima dell'albero del file system. Infine, il resto del disco contiene tutte le altre directory ed i file.

### 6.3.2 Implementazione dei file

Il fattore chiave nell'implementazione della memorizzazione dei file è tener traccia di quali blocchi del disco associare a ciascun file. Si usano vari metodi in sistemi differenti; in questo paragrafo se ne esamineranno alcuni.

#### Allocazione contigua

Lo schema di allocazione più semplice è quello di memorizzare ogni file come un blocco contiguo di dati sul disco; su un disco con blocchi da 1KB, un file di 50KB sarà allocato in 50 blocchi consecutivi, con blocchi da 2KB, sarà allocato in 25 blocchi.

In Figura 6.12(a) vediamo un esempio di allocazione contigua: vengono mostrati i primi 40 blocchi di disco, a partire dal blocco 0, sulla sinistra. Inizialmente, il disco era vuoto, poi un file A, lungo quattro blocchi, è stato scritto sul disco a partire dall'inizio (blocco 0); dopo di ciò un file lungo sei blocchi, B, è stato scritto a partire dalla fine di A. Si noti che ogni file inizia all'inizio di un blocco nuovo, quindi se A fosse stato lungo 3 blocchi e mezzo, si sarebbe perso dello spazio alla fine dell'ultimo blocco. In figura, sono mostrati un totale di 7 file, ognuno dei quali inizia all'inizio di un blocco, e segue il file che era stato precedentemente scritto. Si usano toni di grigio diversi, semplicemente per distinguere meglio da un file all'altro.

L'allocazione contigua dello spazio su disco ha due vantaggi significativi. Primo, è

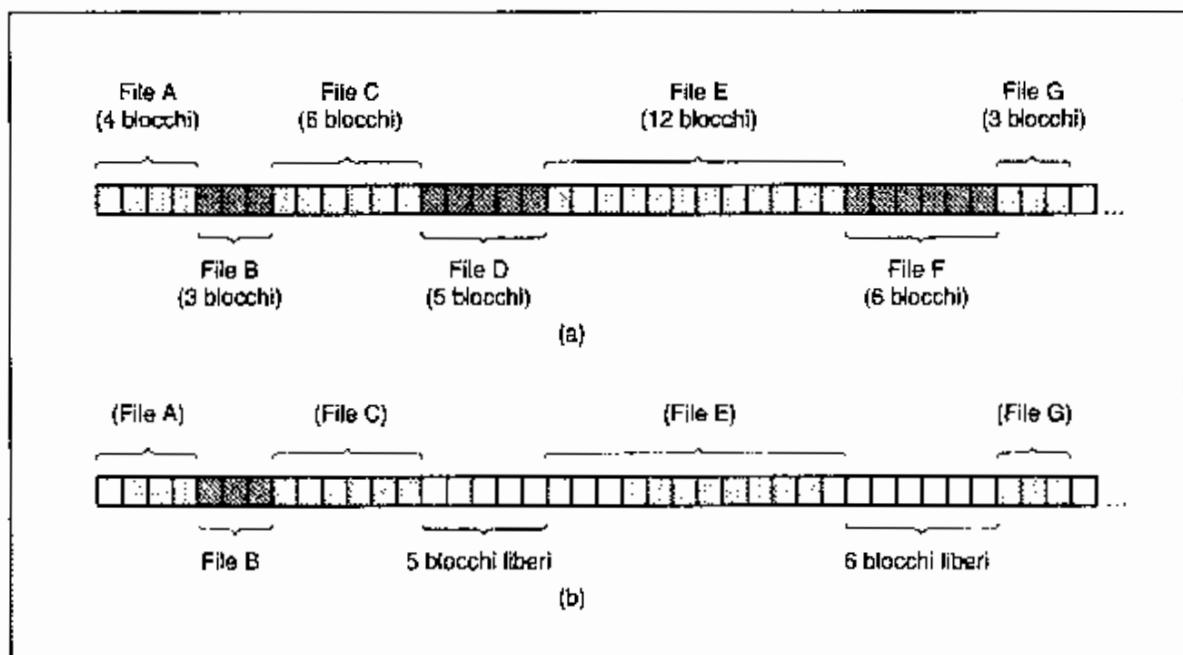


Figura 6.12 [a] Allocazione contigua dello spazio di disco per sette file. [b] Lo stato del disco dopo che i file D e F sono stati rimossi.

semplice da implementare, perché per tener traccia di dove sono i blocchi di un file basta memorizzare due numeri, cioè l'indirizzo su disco del primo blocco e il numero di blocchi del file; dato il numero del primo blocco, il numero di ogni altro blocco può essere calcolato con una semplice addizione.

Secondo, è molto efficiente in lettura, poiché l'intero file si può leggere dal disco con una singola operazione; è necessaria solo un'operazione di posizionamento (sul primo blocco) e, dopo di ciò, non saranno necessari altri posizionamenti e ritardi di rotazione, quindi si può accedere ai dati alla massima velocità di accesso fornita dal disco. Quindi, l'allocazione continua è semplice da implementare e dà alte prestazioni.

Sfortunatamente l'allocazione contigua ha anche un inconveniente ugualmente significativo: col passare del tempo il disco diventa frammentato. Per vedere cosa succede, consideriamo la Figura 6.12(b); sono stati rimossi due file, D e F. Quando si rimuove un file, i suoi blocchi vengono liberati quindi rimangono dei gruppi di blocchi liberi sul disco; il disco non viene compattato subito in modo da eliminare i buchi, perché questo significherebbe copiare tutti i blocchi che seguono il buco, potenzialmente milioni di blocchi; il risultato è che il disco si compone di file e buchi, come illustrato in figura.

Inizialmente la frammentazione non è un problema in quanto ogni nuovo file può essere scritto in fondo al disco, a seguito del precedente. Comunque, ad un certo punto il disco sarà pieno, e si renderà necessario compattarlo, cosa molto costosa, oppure riusare lo spazio libero occupato dai buchi; riusare lo spazio richiede il mantenimento di una lista di buchi, la qual cosa è fattibile. Però, quando viene creato un nuovo file, occorre conoscere la sua dimensione finale, così da scegliere un buco della dimensione giusta per ospitarlo.

Immaginiamo le conseguenze di un simile progetto: l'utente avvia un editore o un word processor e inizia a scrivere un documento, e la prima cosa che il programma chiede è quanti byte diventerà lungo tale documento; rispondere alla domanda è obbligatorio, se si vuole che il programma continui. Inoltre se il numero dato è troppo piccolo, il programma terminerebbe prematuramente perché il buco scelto è troppo piccolo e non c'è spazio per il resto del file. Se l'utente cerca di aggirare il problema rispondendo con numeri incredibilmente grandi, ad esempio 100 MB, l'editor potrebbe non essere in grado di trovare un buco così grande ed annuncerebbe che il file non può essere creato. Naturalmente, l'utente sarebbe libero di ritentare con un numero più piccolo, 50 MB ad esempio, e così via finché non trova un buco sufficiente, ma sicuramente questo schema non produrrebbe molti utenti felici.

Ci sono comunque situazioni in cui l'allocazione contigua è fattibile ed infatti ampiamente usata: sui CD-ROM. In questo caso tutte le dimensioni dei file sono note in anticipo e non cambieranno più. Studieremo il più comune file system per CD-ROM più avanti in questo capitolo.

Come citato nel Capitolo 1, nell'informatica spesso la storia si ripete, via via che nuove generazioni di tecnologia si susseguono; l'allocazione contigua veniva usata su file system di dischi magnetici anni fa, in quanto era semplice e forniva alte prestazioni (a quei tempi la facilità d'uso non era un requisito molto importante); in seguito l'idea fu abbandonata a causa della difficoltà di conoscere la dimensione finale di un file all'atto della sua creazione. Ma, con l'avvento di CD-ROM, DVD e altri mezzi ottici ad una sola scrittura, improvvisamente i file contigui sono ritornati ad essere una buona idea. Quindi è importante studiare i vecchi sistemi e le vecchie idee che erano concettualmente semplici e lineari, in quanto potrebbero essere utili su sistemi futuri, nei modi più sorprendenti.

### Allocazione a lista concatenata

Il secondo metodo per la memorizzazione di file è quello di mantenere ognuno come una lista concatenata di blocchi del disco, come mostrato in Figura 6.13. La prima parola

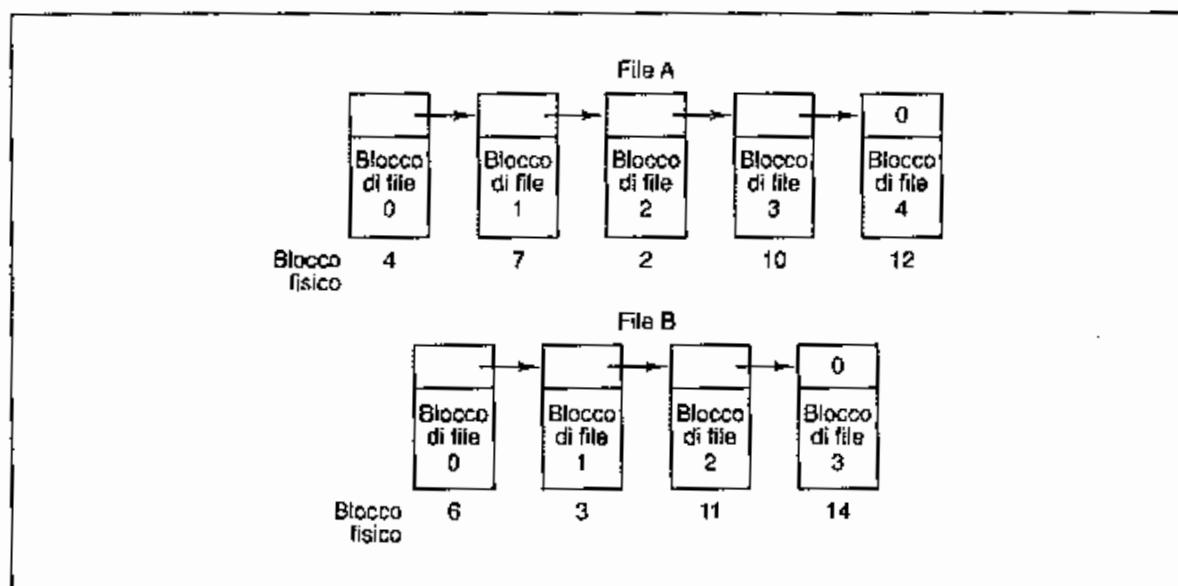


Figura 6.13 Memorizzare un file come una lista concatenata di blocchi di disco.

di ogni blocco è usata come un puntatore al prossimo blocco; la parte rimanente del blocco si utilizza per i dati.

Diversamente dall'allocazione contigua, con questo metodo si può usare ogni blocco di disco; non si perde spazio per la frammentazione del disco (ad eccezione della frammentazione interna nell'ultimo blocco). Inoltre, è sufficiente memorizzare in ogni elemento della directory solamente l'indirizzo su disco del primo blocco, la parte rimanente si può trovare partendo da esso.

D'altra parte, sebbene la lettura sequenziale del file sia semplice, l'accesso diretto è estremamente lento: per arrivare al blocco  $n$  il sistema operativo deve partire dall'inizio e leggere gli  $n - 1$  blocchi precedenti, uno alla volta. Chiaramente, tutte queste operazioni di lettura sono estremamente lente.

Inoltre, la quantità di memoria dati in un blocco non è più una potenza di due poiché il puntatore occupa alcuni byte; pur non essendo critico, avere una dimensione bizzarra è inefficiente, poiché molti programmi leggono e scrivono blocchi la cui grandezza è una potenza di due. Con i primi byte di ogni blocco occupati dal puntatore al blocco successivo, letture dell'intero blocco richiedono l'acquisizione e la concatenazione di informazioni da due blocchi di disco, il che genera ulteriore overhead dovuto alla copiatura.

### Allocazione a lista concatenata usando una tabella in memoria

Entrambi gli svantaggi dell'allocazione a lista concatenata si possono eliminare togliendo i puntatori dai blocchi del disco e mantenendoli in una tabella in memoria. La Figura 6.14 mostra la tabella dell'esempio di Figura 6.13: in entrambe le figure vi sono due file: il file A usa i blocchi di disco 4, 7, 2, 10 e 12, in quest'ordine, e il file B usa nell'ordine i blocchi di disco 6, 3, 11 e 14. Usando la tabella di Figura 6.14 si può partire con il blocco 4 e seguire la catena lungo tutto il percorso fino alla fine; la stessa cosa si può fare partendo con il blocco 6. Le catene terminano con una marcatura speciale (ad esempio con un -1) ossia un simbolo che non sia un numero di blocco valido. Una tabella di questo tipo, tenuta in memoria, viene chiamata **FAT** (*tabella di allocazione di file, File Allocation Table*).

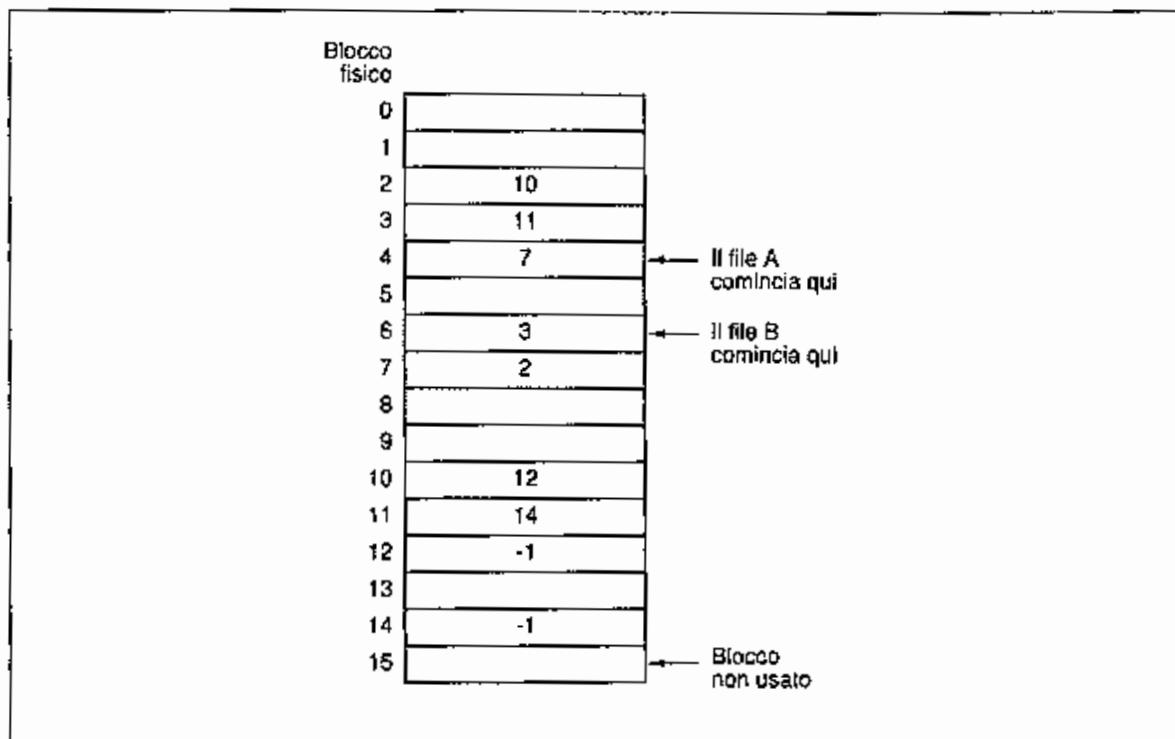


Figura 6.14 Allocazione a lista collegata usando una tabella di allocazione di file in memoria centrale.

Usando questa organizzazione, l'intero blocco è disponibile per i dati. Inoltre l'accesso diretto è molto più facile: sebbene si debba ancora seguire la catena per trovare un dato blocco del file, la catena è interamente in memoria, per cui la si può seguire senza far alcun riferimento al disco. Come il metodo precedente, basta che ogni elemento di directory mantenga un solo intero (il numero del primo blocco) per localizzare tutti i blocchi indipendentemente dalla dimensione del file.

Lo svantaggio principale di questo metodo è che l'intera tabella deve essere sempre in memoria. Con un disco da 20 GB e blocchi da 1 KB, la tabella avrà 20 milioni di elementi, uno per ognuno dei 20 milioni di blocchi del disco, ognuno dei quali dovrà essere di almeno 3 byte. Per aumentare la velocità della ricerca, essi dovranno essere di 4 byte, cioè la tabella occuperà continuamente 60 MB o 80 MB di memoria, a seconda che il sistema sia ottimizzato rispetto al tempo o allo spazio. La tabella potrebbe essere messa in memoria paginabile, ma anche in questo caso occuperebbe una grande quantità di memoria virtuale e di spazio di disco, oltre al fatto di generare ulteriore traffico di paginazione.

## Gli i-node

Il quarto e ultimo metodo per tener traccia dei blocchi appartenenti a un file consiste nell'associare a ogni file una piccola struttura dati detta **i-node** (index node, nodo indice), che elenca gli attributi e gli indirizzi del disco dei blocchi del file; un semplice esempio è mostrato in Figura 6.15. A partire dall'i-node è possibile trovare tutti i blocchi del file; il grande vantaggio di questo schema rispetto all'usare una tabella che risiede in memoria è che l'i-node ha bisogno di stare in memoria solo quando il file ad esso corrispondente viene aperto. Se ogni i-node occupa  $n$  byte, e se possono essere aperti un mas-

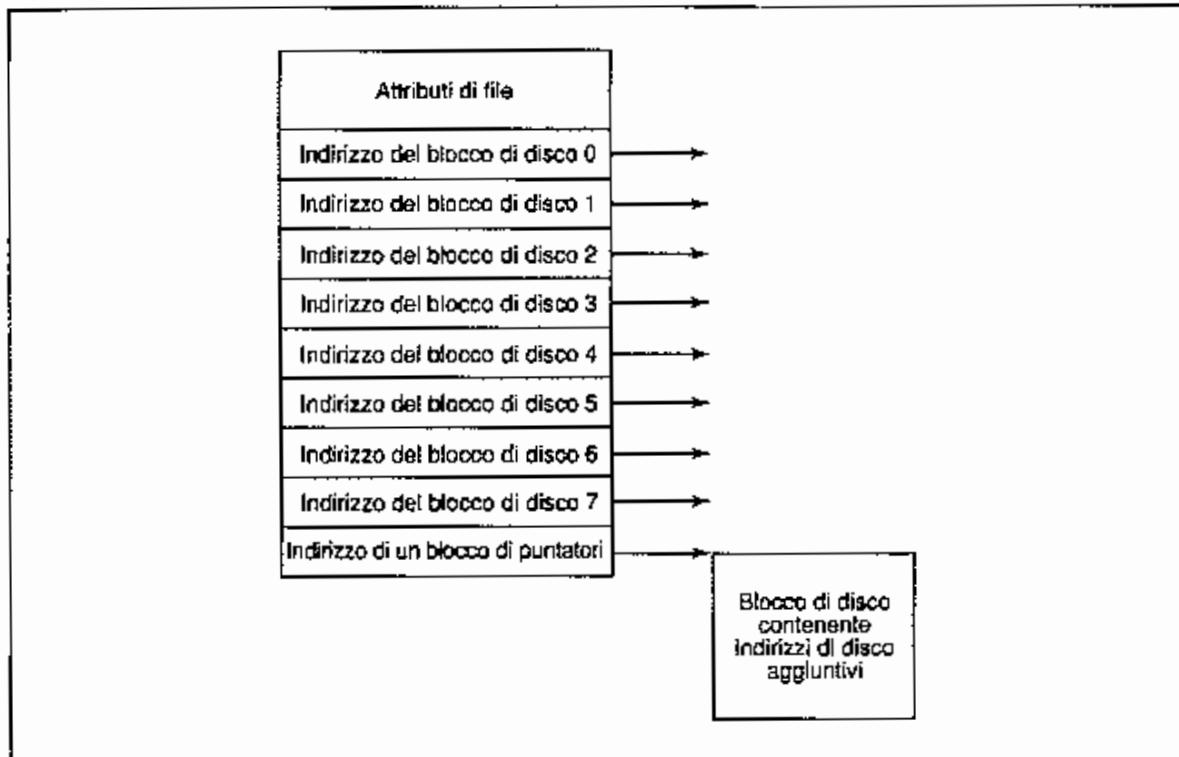


Figura 6.15. Un esempio di i-node

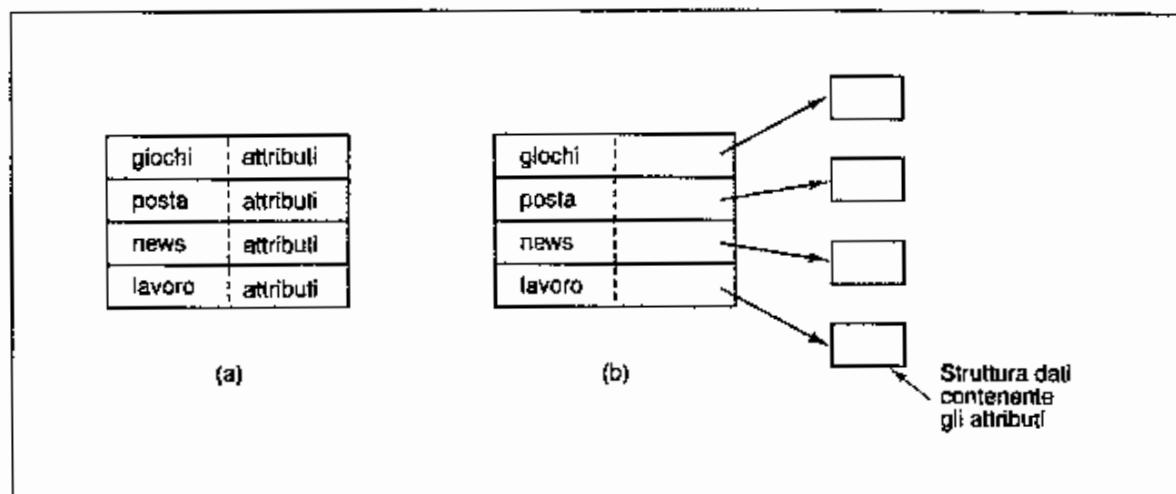
simo di  $k$  file alla volta, la memoria totale occupata dall'array che tiene gli i-node dei file aperti è grande soltanto  $kn$ , e solo questo spazio dovrà essere riservato.

Questo array è solitamente molto più piccolo della tabella descritta nella sezione precedente; la ragione è semplice: la tabella che conserva la lista di tutti i blocchi di disco è proporzionale alla dimensione del disco stesso, se il disco ha  $n$  blocchi la tabella avrà  $n$  ingressi, e se il disco diventa più grande, la tabella cresce linearmente con lui. Invece, lo schema degli i-node richiede un array in memoria la cui dimensione è proporzionale al massimo numero di file che possono essere aperti contemporaneamente, e non importa se la dimensione del disco è di 1 GB o 10 GB o 100 GB.

Un problema con gli i-node è che se ognuno di essi ha spazio per un numero fissato di indirizzi di disco, cosa succede se un file cresce al di là di questo limite? Una soluzione consiste nel riservare l'ultimo indirizzo, non per un blocco dati, ma per l'indirizzo di un blocco contenente altri indirizzi di disco, come mostrato in Figura 6.15. Ancora più avanzata sarebbe la presenza di due o più blocchi di questo tipo, o addirittura di blocchi che puntano a blocchi pieni di indirizzi. Torneremo agli i-node in seguito, studiando UNIX.

### 6.3.3 Implementazione delle directory

Prima che si possa leggerlo, bisogna aprire un file. Quando si apre un file, il sistema operativo usa il path name definito dall'utente per localizzare l'elemento della directory, e questo fornisce le informazioni necessarie per trovare i blocchi del disco. A seconda del sistema questa informazione può essere l'indirizzo del disco dell'intero file (allocazione contigua), il numero del primo blocco (per entrambi gli schemi a liste collegate), o il



**Figura 6.16** (a) Una semplice directory contenente elementi di dimensione fissata con indirizzi di disco e attributi mantenuti all'interno dell'elemento di directory stesso. (b) Una directory in cui ogni elemento semplicemente si riferisce ad un i-node.

numero dell'i-node. In tutti i casi, la funzione principale del sistema di directory è di mappare il nome ASCII del file nell'informazione necessaria per localizzare i dati.

Un problema strettamente collegato è il posto in cui si dovranno memorizzare gli attributi; ogni file system mantiene gli attributi dei file, ad esempio il proprietario del file e il tempo di creazione, e questi devono essere conservati da qualche parte. Una possibilità ovvia è quella di memorizzarli direttamente nell'elemento di directory e ciò è realizzato in molti sistemi: questa opzione è descritta in Figura 6.16(a). In questo semplice schema, una directory si compone di una lista di elementi di dimensione fissata, uno per file, che contengono un nome di file (di lunghezza fissata), una struttura degli attributi del file e uno o più indirizzi di disco (fino ad un massimo) che indicano la posizione dei blocchi di disco.

Per i sistemi che usano i-node, un'altra possibilità è memorizzare gli attributi nell'i-node piuttosto che nell'elemento di directory. In questo caso, l'elemento della directory può essere più piccolo: solo un nome di file e il numero dell'i-node. Questo approccio è mostrato in Figura 6.16(b). Come si vedrà più avanti, questo metodo presenta certi vantaggi rispetto a inserirli nell'elemento di directory; i due approcci mostrati in Figura 6.16 corrispondono, rispettivamente, a MS-DOS/Windows e UNIX, come vedremo nel seguito di questo capitolo.

Fino ad ora abbiamo ipotizzato che i file abbiano nomi brevi e di lunghezza fissata. In MS-DOS i file hanno una base di 1-8 caratteri e un'estensione opzionale di 1-3 caratteri; in UNIX Versione 7, i nomi dei file erano di 1-14 caratteri, incluse eventuali estensioni. Invece, quasi tutti i moderni sistemi operativi supportano nomi di file più lunghi e di lunghezza variabile. Come si può implementare una cosa del genere?

L'approccio più semplice consiste nel fissare un limite alla lunghezza dei nomi di file, tipicamente 255 caratteri, e usare uno degli schemi di Figura 6.16 riservando 255 caratteri per ogni nome di file. Questo approccio è semplice, ma spreca molto spazio, dato che pochi file avranno nomi così lunghi; per ragioni di efficienza, occorre trovare una nuova struttura.

Un'alternativa è quella di abbandonare l'idea che gli elementi di una directory siano tutti della stessa dimensione. Con questo metodo, ogni elemento contiene una porzione fissa, che tipicamente inizia con la lunghezza dell'elemento stesso, seguita dai dati in formato fisso, che solitamente contengono il proprietario, il tempo di creazione, informa-

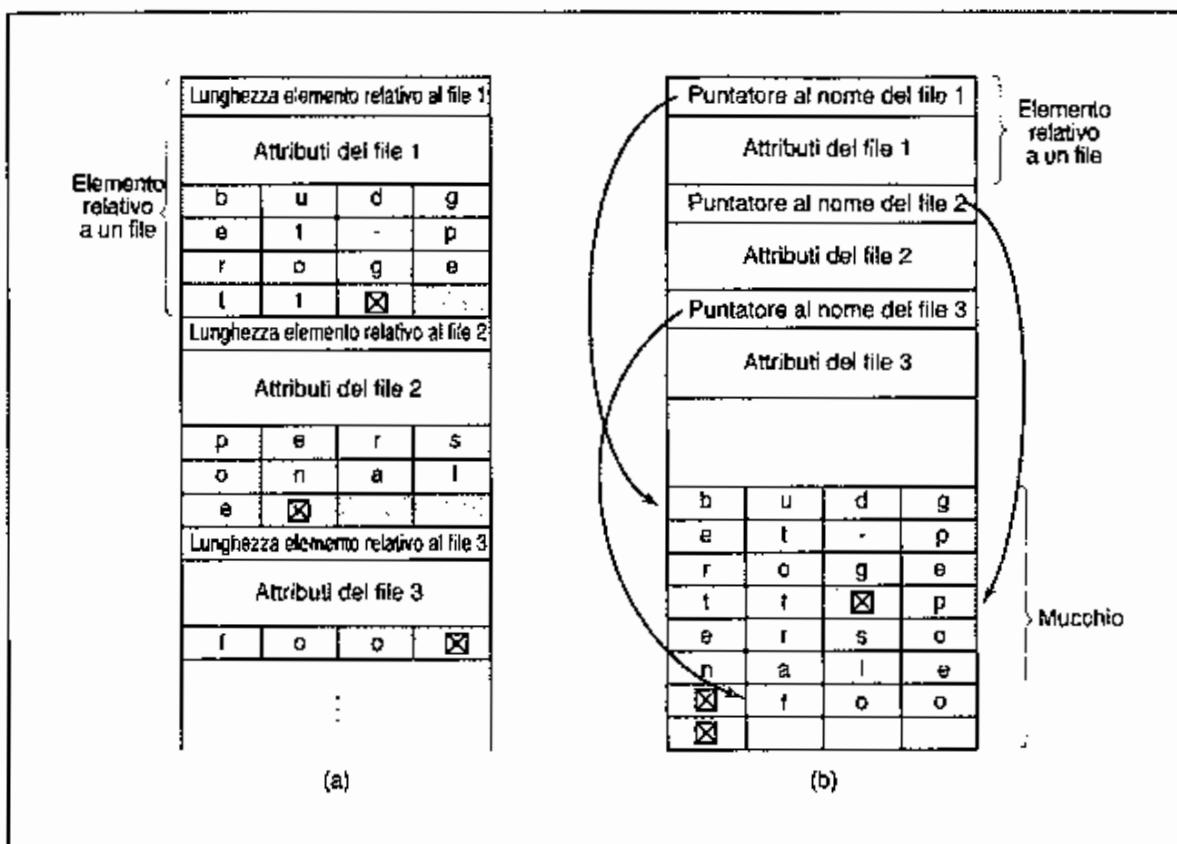


Figura 6.17 Due modi di gestire nomi di file lunghi in una directory. (a) In linea. (b) In un "mucchio".

zioni relative alla protezione e altri attributi. Questa intestazione di lunghezza fissa è seguita del nome di file, di lunghezza qualsiasi, come mostrato in Figura 6.17(a) in formato "big-endian" (ad esempio, delle SPARC). In questo esempio ci sono tre file, *budget\_progett*, *personale* e *foo*; ognuno dei nomi di file termina con un carattere speciale (solitamente uno 0), che è rappresentato in figura da un quadrato con una croce all'interno. Per fare in modo che ogni elemento relativo ad una directory inizi all'estremo di una parola, ogni nome di file è riempito fino a raggiungere un numero intero di parole, come indicato dalle aree grigie nella figura.

Uno svantaggio di questo metodo è che se viene rimosso un file, si introduce uno spazio di lunghezza variabile dentro la directory, e questo spazio può non essere adatto al prossimo file che verrà aggiunto. Questo problema è lo stesso che abbiamo incontrato nel caso di file contigui, anche se in questo caso compattare la directory è un'operazione fattibile perché essa è completamente contenuta in memoria. Un altro problema, invece, consiste nel fatto che un singolo elemento relativo ad una directory può allargarsi su più di una pagina, quindi può verificarsi un page fault durante la lettura di un nome di file.

Un altro modo per gestire nomi di file di lunghezza variabile è quello di avere gli elementi delle directory di dimensione fissa, e lasciare i nomi dei file tutti insieme in un mucchio (heap) alla fine della directory, come mostrato in Figura 6.17(b). Questo metodo ha il vantaggio che, se viene rimosso un elemento, il prossimo file inserito nella directory potrà occupare lo spazio di quello appena eliminato. Ovviamente, il mucchio di nomi di file deve essere gestito, e possono sempre verificarsi dei page fault durante la lettura di un nome di file. Un piccolo dettaglio positivo è che con questa scelta implementativa non

c'è più bisogno di fare iniziare i nomi di file agli estremi delle parole, quindi non saranno necessari caratteri di riempimento in Figura 6.17(b) come invece erano necessari in Figura 6.17(a).

In tutte le configurazioni descritte fino a questo momento, le ricerche di un nome di file avvengono in modo lineare dall'inizio alla fine della directory; per directory estremamente lunghe, la ricerca lineare può essere molto lenta. Un modo per velocizzarla è quello di usare una tabella hash all'interno di ogni directory; se la lunghezza della tabella è  $n$ , per inserire il nome di un file, si deve calcolare partendo da tale nome un valore da 0 a  $n - 1$ , ad esempio dividendolo per  $n$  e prendendo il resto, oppure sommando le parole che compongono il nome, e dividendo la quantità risultante per  $n$ , o qualche operazione simile.

In ogni caso, si ispeziona la componente della tabella corrispondente al codice calcolato; se tale componente è libera, si aggiunge un puntatore all'elemento del file in questione, così gli elementi dei file seguono la tabella di hash. Se la componente è già in uso, si crea una lista collegata, la cui testa è raggiungibile dalla tabella, composta da tutti gli elementi che corrispondono allo stesso valore di hash.

La ricerca di un file segue la stessa procedura; il nome del file viene convertito in numero allo scopo di determinare la componente della tabella hash relativa ad esso, e poi si controllano tutti gli elementi della catena che ha inizio con la componente della tabella stessa. Se il nome non è presente nella catena, allora il file non è presente nella directory.

L'uso delle tabelle di hash ha il vantaggio di consentire ricerche più veloci, ma lo svantaggio di richiedere una gestione molto più complessa; questo metodo può essere preso in considerazione in sistemi dove le directory normalmente contengono centinaia o migliaia di file.

Un modo completamente diverso per velocizzare la ricerca all'interno di directory grandi è quello di mettere in una cache i risultati delle ricerche effettuate; prima di iniziare una ricerca si effettua un controllo per vedere se il nome del file è nella cache, e se c'è, può essere localizzato velocemente, evitando ricerche lunghe. Naturalmente, questo approccio funziona solo se la maggior parte delle ricerche effettuate riguarda un numero relativamente piccolo di file.

### 6.3.4 File condivisi

Quando più utenti stanno lavorando insieme in un progetto, spesso hanno bisogno di condividere i file, di conseguenza per questo tipo di file spesso è conveniente comparire simultaneamente in diverse directory appartenenti a diversi utenti. La Figura 6.18 mostra di nuovo il file system di Figura 6.9, solo che uno dei file di C è presente anche in una delle directory di B. La connessione tra la directory di B e il file condiviso è detta link (collegamento). Il file system stesso è diventato un grafo diretto aciclico o DAG (directed acyclic graph), piuttosto che un albero.

Condividere i file è conveniente ma procura anche qualche problema: per iniziare, se le directory devono realmente contenere gli indirizzi del disco, allora si dovrà fare una copia degli indirizzi del disco nella directory di B quando si collega il file. Se B o C successivamente aggiungono informazioni al file, i nuovi blocchi saranno elencati solo nella directory dell'utente che fa questa operazione; le modifiche non saranno visibili all'altro utente, varificando così lo scopo della condivisione.

Questo problema si può risolvere in due modi: nella prima soluzione i blocchi del disco non sono elencati nelle directory ma in una piccola struttura dati associata al file stesso, e le directory punteranno allora proprio alla struttura dati. In UNIX si usa questo tipo di approccio (in cui la struttura dati è l'i-node).

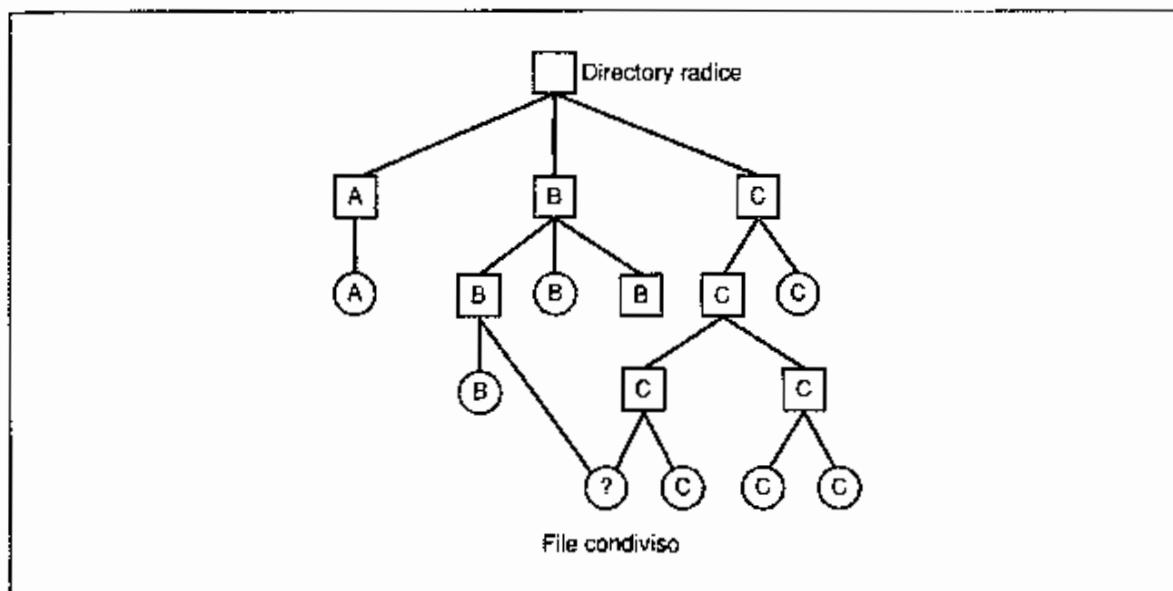


Figura 6.18 File system contenente un file condiviso.

Nella seconda soluzione, B si collega a uno dei file di C per mezzo di un nuovo file, di tipo LINK, creato dal sistema e inserito nella directory di B; tale nuovo file contiene proprio il path name del file a cui è collegato. Quando B legge dal file collegato, il sistema operativo si accorge che il file letto è di tipo LINK, cerca il nome del file e lo legge. Questo approccio è chiamato **collegamento simbolico** o **link simbolico** (symbolic linking).

Ognuno di questi metodi ha i suoi svantaggi: nel primo metodo, nel momento in cui B si collega al file condiviso, l'i-node registra C come proprietario del file. La creazione di un link non cambia il proprietario (vedi Figura 6.19), ma fa incrementare il contatore di link nell'i-node, così il sistema conosce quanti elementi di directory puntano correntemente al file.

Se in seguito C prova a rimuovere il file, il sistema si trova di fronte ad un problema: se rimuove il file e libera l'i-node, B avrà un elemento di directory che punta a un i-node

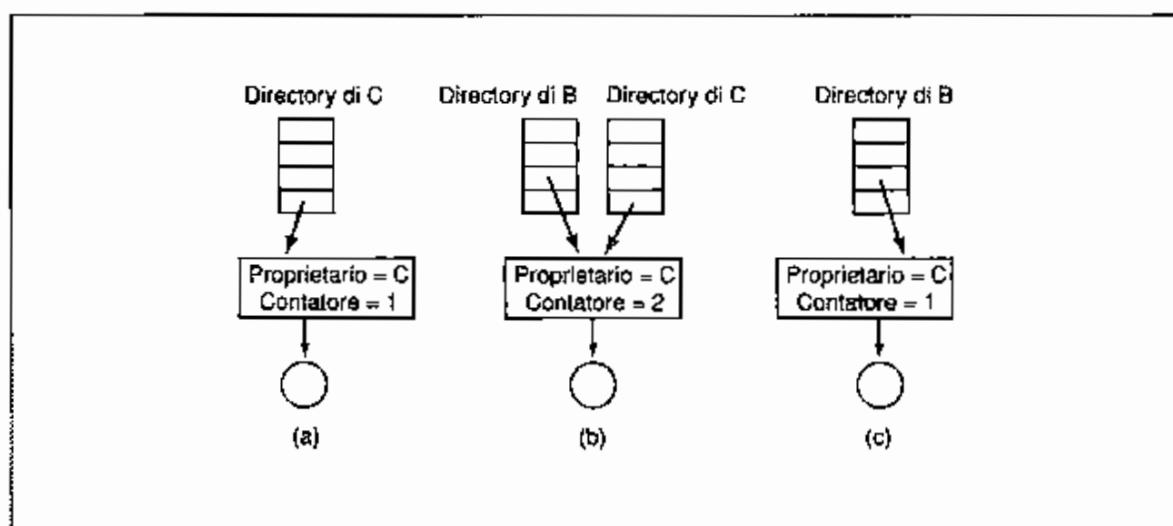


Figura 6.19 (a) Situazione prima del collegamento. (b) Situazione dopo che il collegamento è stato creato. (c) Situazione dopo che il proprietario originale ha rimosso il file.

non valido; e se l'i-node in un secondo momento è riassegnato a un altro file, il link di *B* punterà ad un file sbagliato. Il sistema può vedere dal contatore nell'i-node che il file è ancora in uso, ma non ha modo di trovare tutti gli elementi di directory per il file, per cancellarli: i puntatori alle directory non possono essere memorizzati nell'i-node perché vi può essere un numero illimitato di directory.

La sola cosa da fare è rimuovere l'elemento di directory di *C* ma lasciare l'i-node intatto con il contatore a 1, come mostrato in Figura 6.19 (c): ora si ha una situazione in cui *B* è il solo utente che ha un elemento di directory per un file posseduto da *C*. Se il sistema gestisce l'attribuzione delle risorse o usa il meccanismo delle quote, *C* continuerà a pagare per l'uso del file finché *B* decide di rimuoverlo (ammesso che lo faccia), momento in cui il contatore andrà a zero e il file sarà cancellato.

Con i link simbolici questo problema non sorge perché solo il vero proprietario ha un puntatore all'i-node; gli utenti che hanno collegato il file hanno solo i path name e non i puntatori agli i-node. Quando il proprietario rimuove il file, esso è distrutto, ed in seguito chi proverà a usare il file attraverso un link simbolico fallirà perché il sistema non riesce a trovare il file. Inoltre, la rimozione di un link simbolico non ha alcun effetto sul file.

Il problema con i link simbolici è il sovraccarico aggiuntivo che richiedono: occorre leggere il file che contiene il path, poi analizzarlo e seguirlo lungo ogni componente finché non si raggiunge l'i-node; tutte queste attività possono richiedere un numero considerevole di accessi aggiuntivi al disco. Inoltre per ogni link simbolico si rende necessario un i-node aggiuntivo, ed un blocco aggiuntivo del disco per memorizzare il path, anche se quando il path name è corto, il sistema potrebbe memorizzarlo nello stesso i-node, ottenendo un'ottimizzazione. I link simbolici hanno il vantaggio che si possono usare per collegare file su macchine in qualsiasi parte del mondo, fornendo semplicemente l'indirizzo di rete della macchina su cui il file risiede, ed aggiungendo l'indirizzo della macchina al suo path.

Vi è anche un altro problema introdotto dai link, simbolici o di altro tipo: quando si permettono i link, i file possono avere due o più path. I programmi, che partono da una data directory e trovano tutti i file in quella directory e nelle sue sottodirectory, localizzeranno e agganceranno un file collegato più volte. Per esempio, un programma che scarica tutti i file di una directory e delle sue sottodirectory su un nastro può effettuare più copie di un file raggiungibile tramite un link; inoltre, se il nastro viene riletto in un'altra macchina, e se il programma di dump (scarico) non è stato scritto con attenzione, il file raggiungibile tramite il link sarà copiato due volte sul disco, invece di essere collegato.

### 6.3.5 Gestione dello spazio su disco

I file normalmente sono memorizzati su disco, così la gestione dello spazio su disco è un problema importante per i progettisti di file system. Per memorizzare un file di *n* byte sono possibili due strategie generali: si riservano *n* byte consecutivi dello spazio del disco, oppure il file è diviso in un numero di blocchi (non necessariamente) contigui. Lo stesso compromesso è presente nei sistemi di gestione della memoria tra la segmentazione pura e la paginazione.

Come abbiamo visto, memorizzando un file come una sequenza contigua di byte si presenta l'ovvio problema che se il file aumenta di dimensione, probabilmente dovrà essere spostato nel disco. Lo stesso problema si ha per i segmenti in memoria, con la differenza che muovere un segmento in memoria è un'operazione relativamente veloce rispetto allo spostare un file da una posizione del disco a un'altra; per questa ragione quasi tutti i file system suddividono i file in blocchi di dimensione fissa che non hanno bisogno di essere adiacenti.

## Dimensione dei blocchi

Una volta deciso di memorizzare i file in blocchi di dimensione fissa, occorre decidere quanto dovranno essere grandi i blocchi: considerato il modo in cui sono organizzati i dischi, il settore, la traccia e il cilindro sono ovvi candidati come unità di allocazione (anche se sono tutti dipendenti dal dispositivo, il che non è un vantaggio). In un sistema a pagine, anche la dimensione delle pagine è un candidato possibile.

Avere una grande unità di allocazione, come un cilindro, significa che ogni file, anche di un solo byte, impegna un cilindro intero. Alcune ricerche (Mullender e Tanenbaum, 1984) hanno mostrato che la grandezza media dei file in ambiente UNIX è circa 1K, così per allocare un cilindro di 32K si sprecherà il 31/32 o il 97% dello spazio totale del disco.

D'altra parte, usare una piccola unità di allocazione porterà ogni file a essere composto di molti blocchi. La lettura di ogni blocco normalmente richiede una ricerca e un ritardo di rotazione: quindi la lettura di un file formato da molti piccoli blocchi sarà lenta.

Come esempio si consideri un disco con 131 072 byte per traccia, un tempo di rotazione di 8,33 ms e un tempo medio di ricerca di 10 ms. Il tempo necessario in millisecondi per leggere un blocco di  $k$  byte è quindi la somma del tempo di ricerca, del ritardo di rotazione e del tempo di trasferimento:

$$10 + 4,165 + (k/131072) \times 8,33.$$

La curva continua di Figura 6.20 mostra la velocità di accesso ai dati per un disco del genere come una funzione della dimensione del blocco; per calcolare l'efficienza di spazio, occorre fare un'ipotesi sulla dimensione media dei file; una recente misurazione effettuata presso il dipartimento dell'autore di questo libro, che ospita 1000 utenti e più di un milione di file disco UNIX, dà una mediana di 1680 byte, il che significa che metà dei file sono più piccoli di 1680 byte e l'altra metà più grandi. Occorre notare che la mediana è una metrica migliore della media, in quanto un piccolo numero di file può influenzare enormemente la media, ma non la mediana (la media è infatti di 10845 byte, dovuta in parte alla presenza di alcuni manuali hardware da 100 MB che si trovano on-line). Per semplicità, supponiamo che tutti i file siano di 2 KB, il che produce la curva tratteggiata di Figura 6.20, che rappresenta l'efficienza dello spazio del disco.

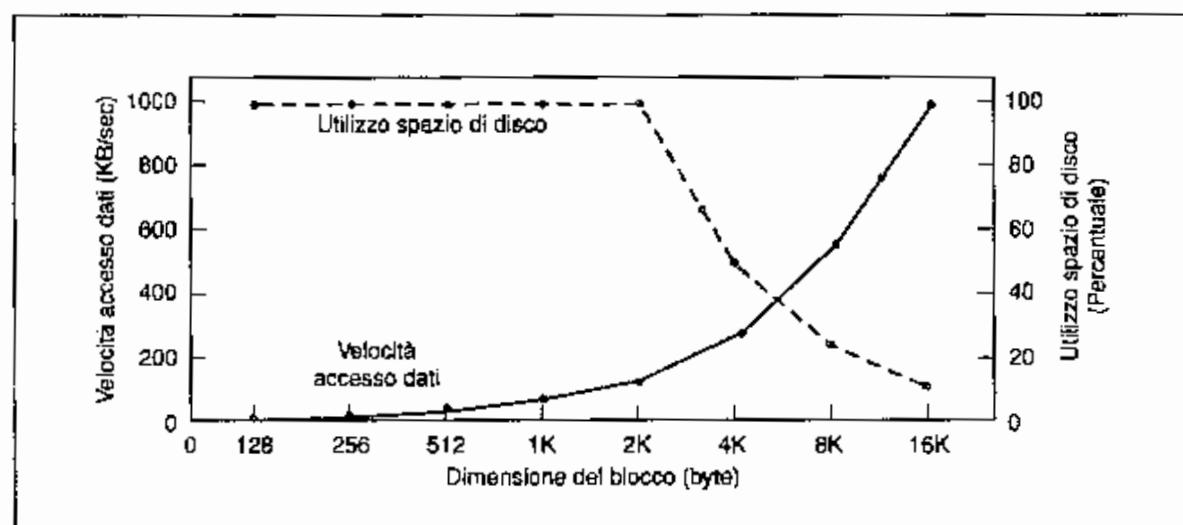


Figura 6.20 La curva continua (la cui scala è indicata sul lato sinistro) fornisce la velocità di accesso ai dati su disco. La curva tratteggiata (la cui scala è indicata sul lato destro) rappresenta l'efficienza dello spazio su disco. Tutti i file sono di 2 KB.

Le due curve possono essere interpretate nel seguente modo: il tempo di accesso ad un blocco è completamente dominato dal tempo di posizionamento e dal ritardo di rotazione, quindi se l'accesso ad un blocco costa 14 ms, più sono i dati recuperati meglio è. Quindi la velocità di accesso ai dati sale al salire della dimensione del blocco (finché il trasferimento non richiede così tanto tempo che il tempo di trasferimento diventa dominante). Con blocchi piccoli che sono potenza di 2 e file di 2 KB, non si ha spreco di spazio in un blocco, ma con file di 2 KB e blocchi di 4 KB o più grandi, si sposta una certa quantità di spazio. In realtà, pochi file occupano un multiplo della dimensione dei blocchi di disco, quindi un po' di spazio viene sempre sprecato alla fine dell'ultimo blocco occupato dal file.

Quello che le curve mostrano, comunque, è che prestazioni e uso dello spazio sono strettamente in conflitto: blocchi piccoli non vanno bene per le prestazioni ma vanno bene per un buon utilizzo dello spazio, quindi una dimensione compromessa è necessaria. Per questi dati, una buona scelta potrebbe essere 4 KB, ma alcuni sistemi operativi fecero le loro scelte molto tempo fa, quando i parametri dei dischi e le dimensioni dei file erano diverse. UNIX usa solitamente 1 KB; i blocchi di disco in MS-DOS possono essere una qualunque potenza di due da 512 byte a 32 KB, ma la loro dimensione è determinata dalla dimensione del disco e legata a ragioni che non hanno nulla a che fare con quanto discusso finora (il massimo numero di blocchi su una partizione di disco è  $2^{16}$ , il che obbliga ad avere blocchi grandi su dischi grandi).

Durante un esperimento fatto allo scopo di verificare se l'uso dei file in Windows NT era diverso dall'uso dei file in UNIX, Vogels effettuò delle misurazioni sui file presso la Cornell University (Vogels, 1999); egli osservò che l'uso dei file in NT è più complicato che in UNIX e scrisse:

*Quando battiamo alcuni caratteri all'interno del text editor BloccoNote, se vogliamo salvare quanto scritto all'interno di un file azioniamo 26 chiamate di sistema, compresi 3 tentativi di apertura di file falliti, 1 sovrascrittura di file e 4 sequenze di open e close aggiuntive.*

Ciononostante egli osservò una dimensione mediana (pesata in base all'uso) dei file letti di 1 KB, dei file scritti di 2,3 KB e di file letti e scritti di 4,2 KB. Tenendo conto del fatto che la Cornell University ha più utenti interessati al calcolo scientifico su larga scala, rispetto all'istituzione che ospita l'autore di questo libro, e tenendo conto delle differenze nelle tecniche di misurazione (statiche contro dinamiche) i risultati di Vogels sono abbastanza coerenti con quelli di una dimensione mediana di circa 2 KB.

## Gestione dei blocchi liberi

Una volta scelta una dimensione del blocco, il problema successivo è come tener traccia dei blocchi liberi; i metodi usati comunemente sono due, come mostrato in Figura 6.21. Il primo utilizza una lista concatenata dei blocchi del disco, in cui ogni blocco contiene la quantità massima possibile di numeri di blocchi di disco liberi. Con un blocco di 1K e un numero di blocco del disco di 32 bit, ogni blocco sulla lista libera contiene i numeri di 255 blocchi liberi. Un disco di 16 GB ha bisogno di una lista libera di al massimo 16794 blocchi per contenere tutti i  $2^{24}$  numeri dei blocchi del disco. Spesso per contenere la lista libera si usano i blocchi liberi.

L'altra tecnica di gestione dello spazio libero è la mappa di bit: un disco con  $n$  blocchi richiede una mappa di bit con  $n$  bit, in cui i blocchi liberi si rappresentano con 1, ed i blocchi allocati con 0 (o viceversa). Un disco di 16 GB ha  $2^{24}$  blocchi da 1 KB e quindi richiede  $2^{24}$  bit per la mappa, ossia necessita di soli 2048 blocchi. Non è sorprendente che la

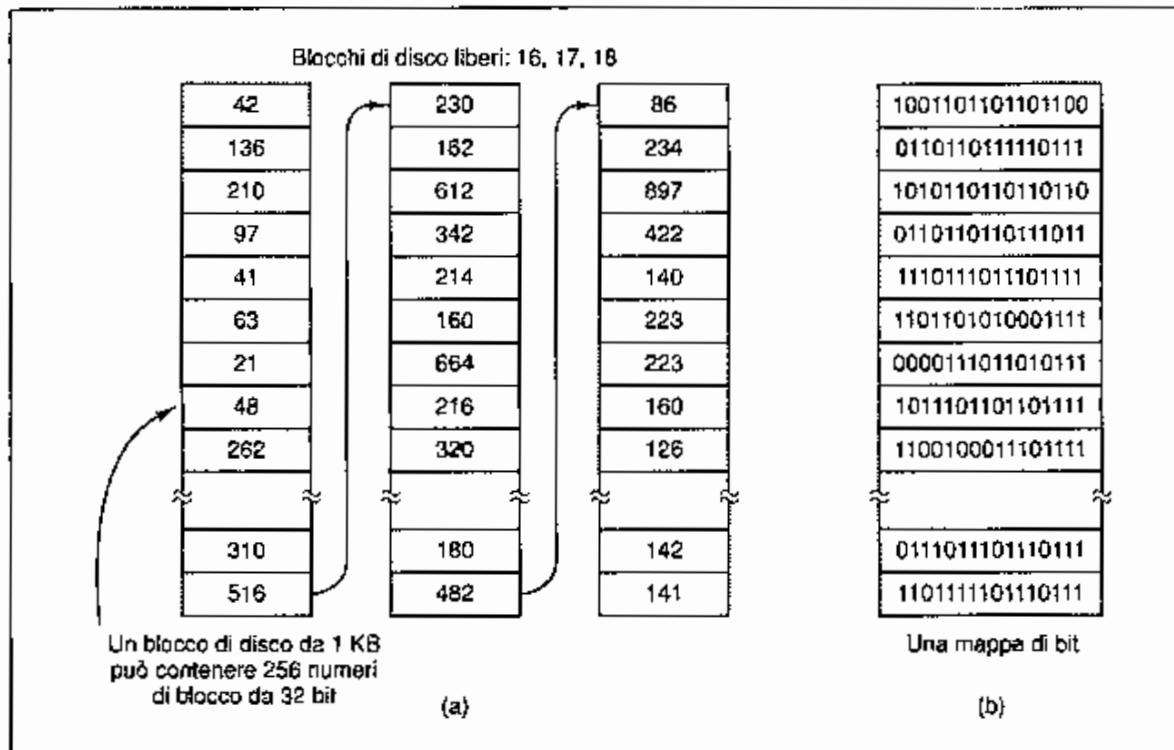


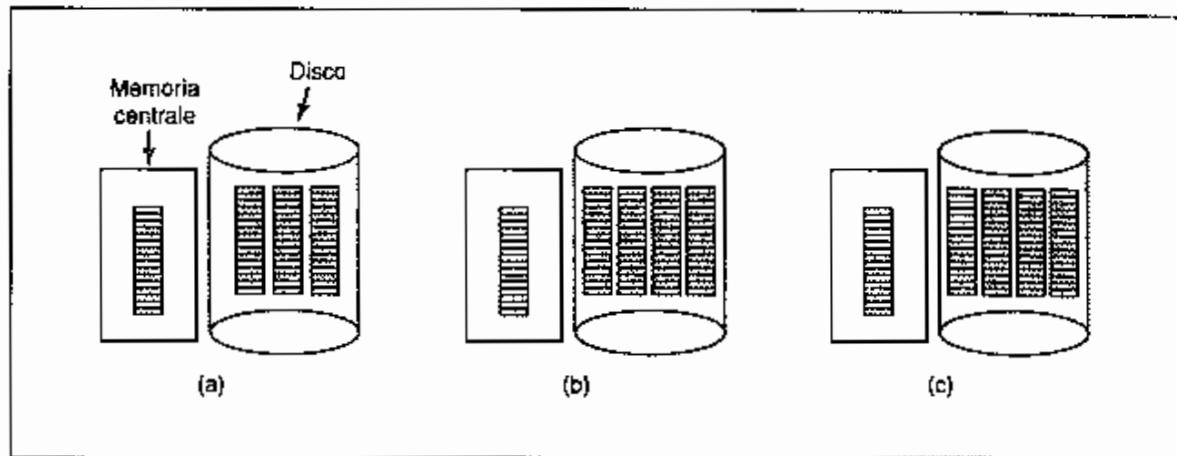
Figura 6.21 (a) Memorizzare la lista libera in una lista concatenata. (b) Usare una mappa di bit.

mappa di bit utilizzi meno spazio poiché essa usa 1 bit per blocco contro i 32 bit del modello a lista concatenata; solo nel caso in cui il disco sia quasi pieno (ossia abbia pochi blocchi liberi) lo schema della lista concatenata richiederà meno blocchi della mappa di bit. D'altra parte, se ci sono molti blocchi liberi, alcuni di essi possono essere chiesti in prestito per ospitare la lista libera, senza perdita di capacità di disco.

Quando viene scelto il metodo della lista libera, solo un blocco di puntatori deve essere mantenuto in memoria centrale: durante la creazione di un file, i blocchi necessari sono presi dal blocco di puntatori; quando il blocco di puntatori si svuota, ne viene letto un altro dal disco. In modo simile, se un file viene cancellato i suoi blocchi sono liberati e aggiunti al blocco di puntatori in memoria centrale e, se il blocco si riempie, esso viene copiato su disco.

In alcune circostanze, questo metodo porta a I/O di disco non necessari: si consideri la situazione di Figura 6.22(a), dove un blocco di puntatori in memoria ha posto per altri due blocchi. Se si deve cancellare un file di tre blocchi, il blocco dei puntatori si completa e quindi deve essere scritto su disco, producendo la situazione descritta in Figura 6.22(b). Se ora si deve scrivere un file di tre blocchi, l'intero blocco di puntatori deve venire riletto dal disco, riportandoci alla situazione di Figura 6.22(a). Se il file di tre blocchi appena scritto era un file temporaneo, quando viene liberato occorre un'altra scrittura su disco per scrivere il blocco dei puntatori ancora una volta su disco. In breve, quando un blocco di puntatori è quasi vuoto, una serie di file temporanei dalla vita breve, possono causare molti I/O sul disco.

Un approccio alternativo che impedisce la maggior parte degli I/O descritti è quello di dividere il blocco di puntatori; quindi, quando tre blocchi sono liberati, invece di andare dalla situazione di Figura 6.22(a) a quella della Figura 6.22(b), andiamo dalla Figura 6.22(a) alla Figura 6.22(c); in questo modo il sistema può gestire una serie di file temporanei senza effettuare alcun I/O di disco. Se il blocco in memoria si riempie, viene scrit-



**Figura 6.22** (a) Un blocco di puntatori a blocchi liberi di disco quasi pieno è in memoria, e tre blocchi di puntatori sono sul disco. (b) Risultato della liberazione di un file di 3 blocchi. (c) Una strategia alternativa per gestire i tre blocchi liberi. Gli elementi in grigio rappresentano puntatori a blocchi di disco liberi.

to sul disco, mentre il blocco mezzo pieno viene copiato dal disco in memoria. L'idea è quella di tenere la maggior parte dei blocchi pieni su disco (per minimizzare l'uso del disco), e tenere un blocco parzialmente vuoto in memoria, in modo da gestire sia la creazione sia la cancellazione di file senza troppi I/O sulla lista libera.

Anche con la mappa di bit è possibile tenere solo un blocco in memoria, cercando un altro blocco su disco solo quando quello presente diventa pieno o vuoto. Un beneficio aggiuntivo di questo approccio è che facendo tutte le allocazioni su un unico blocco della mappa, i blocchi di disco rischiano di essere tutti vicini, e quindi minimizzano le operazioni di posizionamento del braccio del disco. Siccome la mappa di bit è una struttura dati di dimensione fissata, se il kernel è (parzialmente) paginato, allora la mappa di bit può essere tenuta in memoria virtuale e parti di essa possono venire paginate quando necessario.

## Il meccanismo delle quote su disco

Per evitare che gli utenti si impossessino di troppo spazio del disco, i sistemi operativi con molti utenti offrono spesso un meccanismo per impostare quote di uso del disco. L'idea è che l'amministratore del sistema assegna ad ogni utente una quantità massima di file e di blocchi, e il sistema operativo assicura che gli utenti non superino le loro quote. Di seguito si descrive un meccanismo tipico.

Quando un utente apre un file, gli attributi e gli indirizzi del disco sono localizzati e sistemati in una tabella di file aperti in memoria centrale; tra gli attributi vi è una voce che ne indica il proprietario, e tutti gli aumenti nella dimensione del file saranno addebitati alla quota dell'utente proprietario.

Una seconda tabella contiene un record con la quota di ogni utente che ha un file correntemente aperto, anche se il file era stato aperto da qualcun altro. Questa tabella è mostrata in Figura 6.23: essa è estratta da un file di quote su disco per gli utenti i cui file sono correntemente aperti. Quando tutti i file vengono chiusi il record è riscritto nel file di quote.

Quando si crea una nuova voce all'interno della tabella dei file aperti, vi si inserisce un puntatore al record con la quota del proprietario per rendere facile il ritrovamento dei

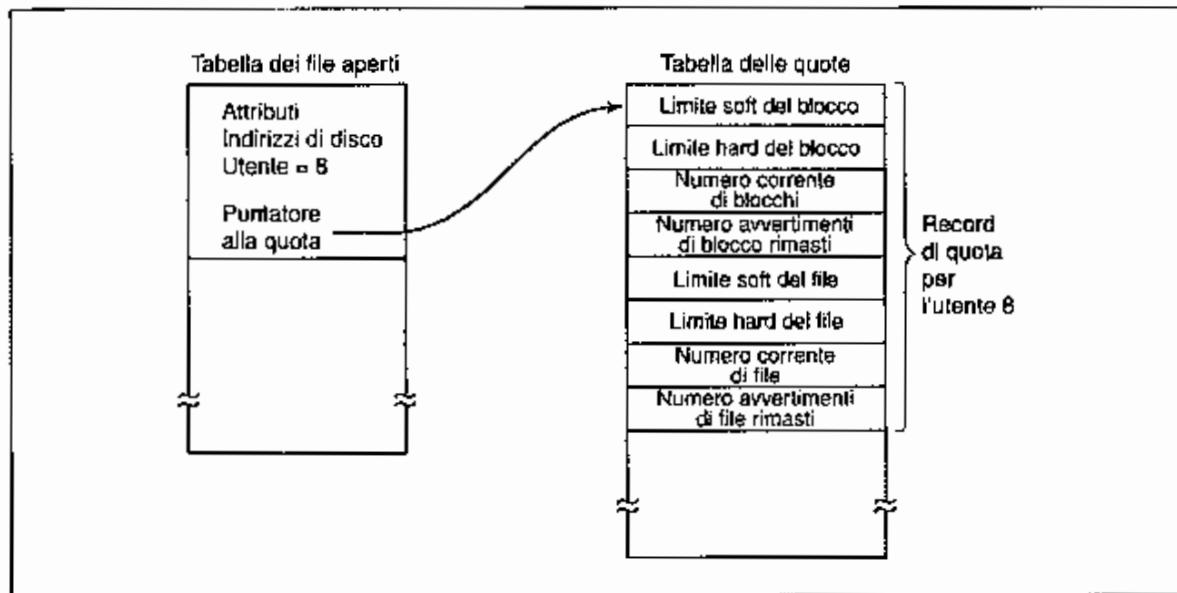


Figura 6.23 Si tiene traccia delle quote, utente per utente, su una tabella delle quote.

valori limite. Ogni volta che si aggiunge un blocco a un file, si incrementa il numero totale di blocchi attribuito al proprietario, e si fa un controllo rispetto a due limiti, chiamati hard (rigido) e soft (variabile). Si può superare il limite soft ma non il limite hard; un tentativo di aggiungere un file quando si è raggiunto il limite hard del blocco provocherà un errore. Controlli analoghi esistono anche per il numero di file.

Quando un utente tenta di entrare, il sistema controlla il file delle quote per verificare se l'utente ha superato il limite soft per il numero di file o di blocchi del disco. Se è stato violato un limite, viene visualizzato un avvertimento e il contatore degli avvertimenti rimanenti è diminuito di uno; se il contatore raggiunge lo zero, l'utente ha ignorato l'avvertimento una volta di troppo, e non gli è permesso di collegarsi. Per riavere il permesso di collegarsi di nuovo dovrà fornire qualche chiarimento all'amministratore del sistema.

Questo metodo ha la caratteristica che gli utenti possono superare i loro limiti soft durante una sessione, purché provvedano a rimuovere gli eccessi prima di chiudere il collegamento; ma i limiti hard non si possono mai superare.

### 6.3.6 Affidabilità del file system

Spesso la distruzione di un file system è un disastro molto più grande della distruzione di un computer: se un computer viene distrutto dal fuoco, colpito da un fulmine, o se una tazza di caffè si versa sulla tastiera, sarà un guaio e costerà denaro, ma generalmente si può comprare un ricambio con un minimo di difficoltà. Personal computer economici si possono anche rimpiazzare in poche ore andando semplicemente dal venditore (eccetto che all'università, dove fare un acquisto richiede tre comitati, cinque firme e 90 giorni).

Se il file system di un computer è irrevocabilmente perduto, per problemi di hardware, software o per floppy disk rosicchiati da topi, ripristinare tutte le informazioni sarà difficile e comporterà una grande perdita di tempo, e in molti casi sarà impossibile. Per le persone i cui programmi, documenti, file clienti, dichiarazioni delle tasse, basi di dati, piani di mercato o altre informazioni sono perdute per sempre le conseguenze possono

essere catastrofiche. Mentre il file system non può offrire alcuna protezione dalla distruzione fisica delle apparecchiature e dei supporti fisici, esso può aiutare a proteggere l'informazione; in questo paragrafo si tratteranno alcuni argomenti relativi alla salvaguardia del file system.

I dischetti sono solitamente perfetti quando lasciano la fabbrica, ma possono prodursi blocchi rovinati durante l'uso; i dischi fissi hanno solitamente blocchi rovinati sin dall'inizio: è semplicemente troppo costoso produrre solo dischi fissi privi di errori. Come abbiamo visto nel Capitolo 5, i blocchi rovinati sono generalmente gestiti dal controllore, il quale li sostituisce con settori di ricambio pensati per tale scopo. A prescindere da queste tecniche, ci sono anche altre questioni relative all'affidabilità che affronteremo qui di seguito.

## Copie di backup

La maggior parte delle persone non crede che fare delle copie di backup dei loro file valga la spesa ed il tempo perso – finché un bel giorno il loro disco improvvisamente muore, e la maggior parte di essi si pente sul letto di morte. Invece le società (solitamente) ben comprendono il valore dei loro dati, e quindi fanno copie di backup, almeno una volta al giorno, solitamente su nastro: i nastri moderni contengono decine o anche centinaia di gigabyte e costano pochissimo per gigabyte. Ciononostante fare backup non è così facile come sembra, quindi nel seguito esamineremo le questioni più importanti relative a tali operazioni.

I backup su nastro vengono solitamente fatti per risolvere due problemi potenziali:

1. Il ripristino da disastri.
2. Il ripristino da stupidità.

Il primo caso copre situazioni in cui il computer debba essere ripristinato dopo un crash di disco, un incendio, un'inondazione o altre catastrofi naturali; nella pratica queste cose non succedono molto spesso ed è per questo che molta gente non si disturba a fare backup. Le stesse persone tendono a non avere un'assicurazione contro gli incendi della loro casa per lo stesso motivo.

Il secondo caso ha a che fare col fatto che spesso le persone rimuovono accidentalmente dei file dei quali in seguito avranno bisogno; questo problema è così frequente che in Windows, quando un file viene "rimosso" esso non viene affatto cancellato, ma viene messo in una directory speciale chiamata *cestino dei rifiuti*, dalla quale può essere ripescato e ripristinato facilmente. I backup prendono questo principio e lo estendono, permettendo che file che erano stati rimossi giorni o anche settimane fa, possano essere ripristinati da vecchi nastri di backup.

Fare backup richiede molto tempo e occupa grandi quantità di spazio, quindi è importante farlo in modo efficiente e comodo. Queste considerazioni sollevano i seguenti problemi: prima di tutto, è meglio fare il backup dell'intero file system o solo di una parte di esso? In molte installazioni i programmi eseguibili (binari) sono mantenuti in una parte circoscritta dell'albero del file system; non è necessario fare il backup di questi file, in quanto possono essere reinstallati a partire dai CD-ROM forniti dai produttori. Inoltre, la maggior parte dei sistemi hanno delle directory per i file temporanei; solitamente non c'è alcun motivo per fare il backup di questi file. In UNIX, tutti i file speciali (i dispositivi di I/O) sono conservati nella directory */dev*; non solo il backup di questa directory non è necessario, ma è addirittura dannoso in quanto il programma di backup potrebbe rimanere bloccato all'infinito se decidesse di leggere ognuno di questi file fino al completa-

mento. In breve, solitamente è consigliabile fare il backup solo di directory specifiche e di tutto il loro contenuto, piuttosto che dell'intero file system.

In secondo luogo, è una perdita di tempo fare il backup di file che non sono cambiati dall'ultimo backup, il che conduce all'idea dei **dump incrementali**. La forma più semplice di dump incrementale è fare periodicamente un dump (backup) completo, ad esempio settimanalmente o mensilmente, e fare un dump giornaliero solo di quei file che sono stati modificati dall'ultimo dump completo. Uno schema migliore è fare ogni volta il dump solo di quei file che sono stati cambiati dall'ultima backup: questo schema, che minimizza il tempo di backup, rende l'operazione di recupero più complicata in quanto prima di tutto occorre ripristinare il dump completo più recente, ed in seguito tutti i dump incrementali in ordine inverso; per facilitare il recupero sono spesso usati schemi di dump incrementale più sofisticati.

Terzo, siccome il dump solitamente coinvolge immense quantità di dati, può essere necessario comprimerli prima di copiarli sul nastro; comunque, con molti programmi di compressione, un singolo punto rovinato sul nastro di backup può confondere l'algoritmo di decompressione e rendere l'intero file o addirittura l'intero nastro illeggibile; di conseguenza la decisione di comprimere il flusso di dati di backup deve essere considerata con estrema attenzione.

Quarto, è difficile effettuare il backup di file system attivi; se vengono aggiunti, cancellati o modificati file e directory durante il processo di dump, il risultato può essere inconsistente. Al tempo stesso, siccome l'operazione di dump può richiedere ore, può essere necessario portare il sistema off-line per una notte, ad esempio, il che non sempre è accettabile. Per questo motivo, sono stati pensati algoritmi che producono rapidamente delle fotografie dello stato dei file system, copiando le strutture dati critiche, e quindi facciano sì che eventuali cambiamenti di file e directory producano copie dei blocchi invece di cambiamenti sul posto (Hutchinson et al., 1999); in questo modo, il file system è congelato al momento della fotografia e può essere copiato comodamente in seguito.

Quinta e ultima considerazione, fare backup introduce molti problemi non tecnici all'interno di una organizzazione: il miglior sistema di sorveglianza on-line del mondo può diventare inutile se l'amministratore di sistema tiene i nastri di backup nel suo ufficio e lascia l'ufficio aperto e incustodito ogni volta che esce per andare a recuperare una stampa; quello che una spia deve fare è entrare per un secondo, infilarsi un nastro in tasca e tagliare la corda: e addio sicurezza! Inoltre, backup giornalieri servono a poco se l'incendio che distrugge i computer distrugge anche tutti i nastri di backup: per questo motivo, i nastri di backup andrebbero conservati lontano dai computer, ma questo aggiunge altri rischi alla sicurezza. Per una rassegna completa di questi e di altri problemi amministrativi pratici, consultare (Nemeth et al., 2000). Nel seguito ci limiteremo a discutere questioni tecniche riguardanti i backup di file system.

Si possono usare due strategie per fare il dump di un disco su di un nastro: il dump fisico o il dump logico. Il **dump fisico** inizia al blocco 0 del disco, scrive tutti i blocchi di disco sul nastro in ordine, e si ferma quando ha copiato l'ultimo; tale programma è così semplice che può essere scritto, molto probabilmente, in modo che sia completamente privo di bachi, qualcosa che molto probabilmente non si può dire di nessun altro programma utile.

Ciononostante, è necessario fare diversi commenti circa il dump fisico. In primo luogo, non c'è nessuna ragione di copiare blocchi di disco non utilizzati: se il programma di dump riesce ad accedere alla struttura dati dei blocchi liberi, può evitare il dump dei blocchi non usati. Però, saltare alcuni blocchi significa dover scrivere il numero di ogni blocco in cima al blocco stesso (o qualcosa di equivalente), per cui non è più vero che il  $k$ -esimo blocco sul disco equivale al  $k$ -esimo blocco sul nastro.

Un secondo problema riguarda il dump di blocchi danneggiati. Se tutti i blocchi dan-

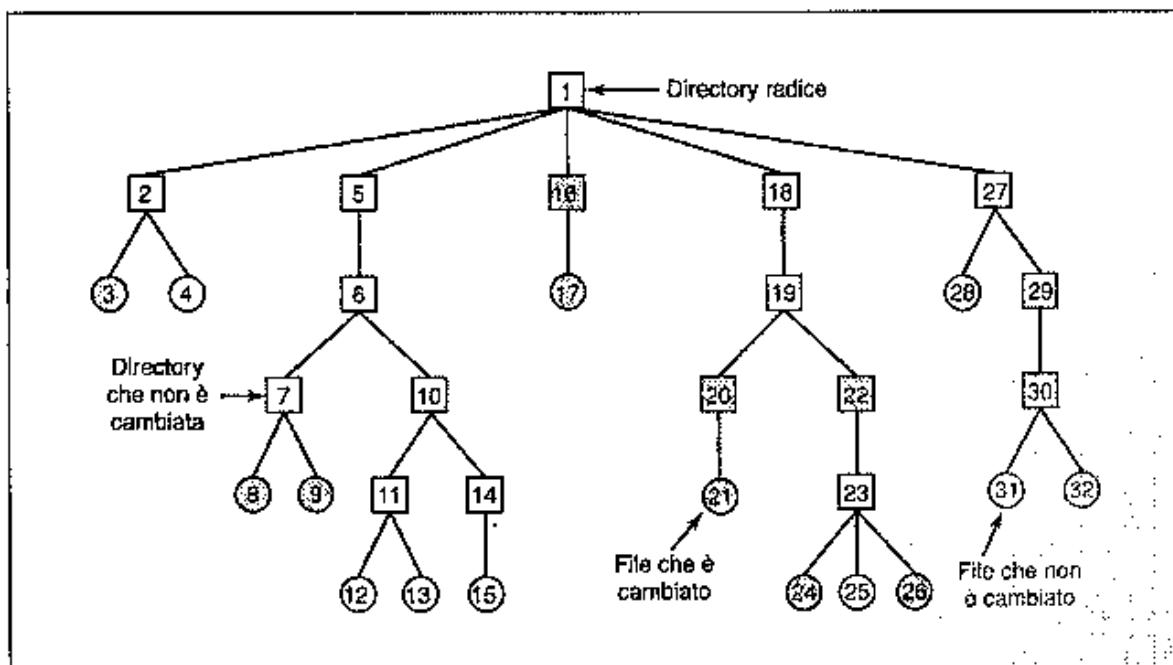
neggiati vengono rimappati dal controllore del disco e nascosti al sistema operativo come descritto nella Sezione 5.4.4, il dump fisico funziona bene; se invece essi sono visibili al sistema operativo e mantenuti in uno o più "file di blocchi danneggiati" o mappe di bit, è assolutamente necessario che il programma che esegue il dump fisico possa accedere a queste informazioni ed eviti di fare il dump di tali blocchi, in modo da non trovarsi infiniti errori di lettura durante l'operazione di dump.

I maggiori vantaggi del dump fisico sono la sua semplicità e l'estrema velocità (essenzialmente può venire eseguito alla velocità del disco); i principali svantaggi sono l'inabilità di saltare directory scelte, fare dump incrementali e ripristinare singoli file su richiesta. Per queste ragioni, la maggior parte dei sistemi usano dump logici.

Un **dump logico** inizia in corrispondenza di una o più directory specificate, e ricorsivamente effettua il dump di tutti i file e le directory che trova che sono cambiate rispetto ad una certa data (ad esempio, l'ultimo backup di un dump incrementale o, per un dump completo, un'installazione di sistema); quindi in un dump logico, il nastro di dump riceve una serie di file e directory accuratamente identificati, il che facilita il ripristino su richiesta di un file specifico o di una directory specifica.

Siccome il dump logico è la forma più frequente, esaminiamo in dettaglio un algoritmo comune, usando l'esempio di Figura 6.24 come guida; la maggior parte dei sistemi UNIX usa questo algoritmo. In figura vediamo un albero di file con directory (indicate da quadrati) e file (cerchi); gli elementi su sfondo grigio sono stati modificati dall'ultima data di riferimento e quindi è necessario scaricarli nuovamente sul nastro, mentre gli elementi bianchi non ne hanno bisogno.

Questo algoritmo scarica tutte le directory (anche quelle non modificate) che giacciono sul cammino di un file o una directory da modificare; questo si fa per due diversi motivi: prima di tutto, per rendere possibile il recupero dei file e delle directory scaricate su un diverso computer con un file system vuoto; in questo modo, i programmi di dump e recupero possono essere usati per trasportare file system tra computer diversi.



**Figura 6.24** Un file system da scaricare sul nastro. I quadrati sono directory e i cerchi file; gli elementi in grigio sono stati modificati dall'ultimo dump. Ogni directory e ogni file sono etichettati con il loro numero di i-node.

La seconda ragione per scaricare tutte le directory è quella di rendere possibile il recupero incrementale di un singolo file (solitamente per gestire il recupero dalla stupidità); supponiamo che un dump di file system completo sia stato effettuato la domenica sera e che un dump incrementale sia effettuato il lunedì sera; martedì viene rimossa la directory `/usr/paolo/progetto/nr3`, insieme a tutti i file e directory al di sotto di essa. Mercoledì mattina, l'utente vuole ripristinare il file `/usr/paolo/progetto/nr3/piani/sommario`, ma non è possibile recuperare unicamente il file `sommario`, perché non c'è il posto dove metterlo: bisogna prima di tutto ripristinare le directory `nr3` e `piani`. Per ottenere tutte le informazioni corrette, quali i proprietari, i modi, i tempi, e così via, queste directory devono essere presenti sul nastro di dump, anche se esse non erano state modificate dall'ultimo dump completo.

L'algoritmo di dump mantiene una mappa di bit indicizzata sui numeri di i-node con diversi bit per i-node. I bit saranno impostati e annullati all'interno della mappa, man mano che l'algoritmo procede, in quattro fasi. La Fase 1 comincia dalla directory di inizio (la directory radice in questo caso), ed esamina tutti gli elementi al suo interno: per ogni file modificato, l'i-node corrispondente viene segnato all'interno della mappa di bit. Anche le directory vengono marcate (che siano state modificate o meno) e in seguito ispezionate ricorsivamente.

Alla fine della Fase 1, tutti i file modificati e tutte le directory sono state marcate nella mappa di bit, come mostrato in Figura 6.25(a) (elementi in grigio). La Fase 2 concettualmente ripercorre l'albero, smarcando tutte le directory che non contengono file o directory modificati al loro interno o sotto di esse. Questa fase lascia la mappa di bit come mostrato in Figura 6.25(b): si noti che le directory 10, 11, 14, 27, 29 e 30 sono state smarcate, perché non contengono nulla al di sotto di esse che sia stato modificato; esse non saranno scaricate. Per contro, le directory 5 e 6 verranno scaricate anche se esse non sono state modificate, perché saranno necessarie per ripristinare i cambiamenti odierni su una macchina nuova. Per ragioni di efficienza, le Fasi 1 e 2 possono essere combinate in un solo attraversamento dell'albero.

A questo punto si sa quali directory e file devono essere scaricati: quelli marcati in grigio in Figura 6.25(b). La Fase 3 consiste nell'analizzare tutti gli i-node in ordine di numero e scaricare tutte le directory marcate, come mostrato in Figura 6.25(c); ogni directory viene preceduta dai suoi attributi (proprietario, tempi, ecc.) così che possa essere ripristinata. Infine, nella Fase 4, i file marcati in Figura 6.25(d) vengono scaricati, anch'essi preceduti dagli attributi; questa fase completa il dump.

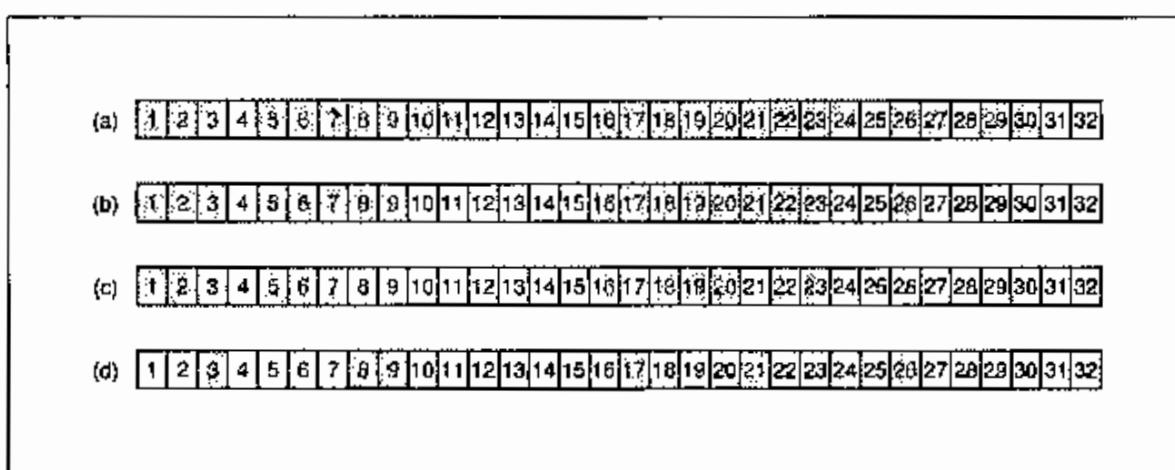


Figura 6.25 Mappe di bit usate dall'algoritmo di dump logico.

Il ripristino di un file system da nastri di dump avviene in modo intuitivo: prima di tutto si crea un file system vuoto sul disco e poi si ripristina il dump completo più recente; siccome le directory appaiono tutte per prime sul nastro, esse vengono ripristinate per prime e forniscono lo scheletro del file system. A questo punto vengono ripristinati i file; questo processo viene ripetuto con il primo dump incrementale fatto dopo il dump completo, poi col successivo e così via.

Nonostante il dump logico sia intuitivo, vi sono alcuni punti delicati da considerare. Prima di tutto, siccome la lista dei blocchi liberi non è un file, non era presente sul nastro e quindi dopo il ripristino deve essere ricostruita al completamento dell'operazione di ripristino; questo è sempre possibile, in quanto i blocchi liberi sono semplicemente il complementare dell'insieme di blocchi utilizzati da tutti i file esistenti.

Un'altra questione sono i link; se un file è collegato a due o più directory, è importante che venga ripristinato un'unica volta e che tutte le directory alla quale era collegato ripristino i giusti puntatori.

Un'altra questione ancora, è che in UNIX i file possono contenere buchi: è legale aprire un file, scrivere qualche byte, posizionarsi su un offset lontano e scrivere qualche altro byte; i blocchi intermedi non sono parte del file e non dovrebbero quindi venire scaricati né ripristinati, ad esempio, i file core hanno spesso ampi buchi tra segmenti di dati e la pila. Se questo fenomeno non viene trattato con attenzione, ogni file di core ripristinato riempie con degli zeri tali aree, e diventa delle stesse dimensioni dello spazio di indirizzamento virtuale (ad esempio  $2^{32}$  o peggio  $2^{64}$  byte).

Infine file speciali, pipe con nome e simili, non dovrebbero mai essere scaricate, non importa in quale directory possano trovarsi (non necessariamente in `/dev`). Per maggiori informazioni circa il backup di file system, consultare (Chervenak et al., 1998 e Zwicky, 1991).

## Consistenza del file system

Un'altra area in cui l'affidabilità costituisce un problema è la coerenza del file system. Molti file system leggono blocchi, li modificano e in un secondo momento li scrivono; se il sistema ha un crash prima che tutti i blocchi modificati siano stati scritti, il file system può trovarsi in uno stato inconsistente. Questo è un problema critico specialmente se alcuni dei blocchi che non sono stati scritti sono blocchi i-node, blocchi directory o blocchi contenenti la lista libera.

Per trattare il problema della inconsistenza del file system, la maggior parte dei computer ha un programma di servizio che controlla la coerenza del file system, ad esempio UNIX ha `fsck`, mentre Windows `Scandisk`. Questi programmi sono attivati ad ogni avvio del sistema, in particolare dopo un crash. La descrizione che segue riguarda `fsck`; `Scandisk` è piuttosto diverso perché funziona su un file system diverso, ma il principio generale di usare la ridondanza intrinseca del file system per verificarne la consistenza è ancora valido. Tutti i controllori di file system verificano ogni file system (ogni partizione di disco) indipendentemente dagli altri.

Si hanno due tipi di controlli di coerenza: per blocchi e per file; per controllare la coerenza dei blocchi il programma costruisce due tabelle ciascuna con un contatore per ogni blocco, entrambi inizialmente a 0. La prima tabella tiene traccia di quante volte il blocco è presente in un file; la seconda memorizza quanto spesso esso è presente nella lista libera (o nella mappa di bit dei blocchi liberi).

Il programma poi legge tutti gli i-node; partendo da un i-node, è possibile costruire la lista di tutti i numeri di blocco usati nel file corrispondente; appena si legge un numero di blocco, si incrementa il suo contatore nella prima tabella. Poi il programma esamina la

| Numero di blocco |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    | Numero di blocco  |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |
|------------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|-------------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|
| 0                | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0                 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |   |
| 1                | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1  | 1  | 1  | 0  | 0  | 1  | Blocchi<br>in uso | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0  | 0  | 1  | 1  | 1  | 0  | 0 |
| 0                | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 1  | 1  | Blocchi<br>liberi | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1  | 1  | 0  | 0  | 0  | 1  | 1 |
| (a)              |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    | (b)               |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |
| 0                | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0                 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |   |
| 1                | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1  | 1  | 1  | 0  | 0  | 1  | Blocchi<br>in uso | 1 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 1 | 0  | 0  | 1  | 1  | 1  | 0  | 0 |
| 0                | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 1 | 1 | 0  | 0  | 0  | 1  | 1  | 0  | Blocchi<br>liberi | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1  | 1  | 0  | 0  | 0  | 1  | 1 |
| (c)              |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    | (d)               |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |

Figura 6.26 Stati del file system. [a] Consistente. [b] Blocco mancante. [c] Blocco duplicato nella lista libera. [d] Blocco dati duplicato.

lista libera o la mappa di bit per trovare tutti i blocchi che non sono in uso; ogni occorrenza di un blocco nella lista libera fa sì che sia incrementato il suo contatore nella seconda tabella.

Se il file system è coerente, ogni blocco avrà un 1 nella prima tabella o nella seconda tabella, come mostrato in Figura 6.26(a). Comunque, come risultato di un crash, le tabelle possono essere tipo quelle di Figura 6.26(b), in cui il blocco 2 non compare in nessuna delle tabelle: si dirà che esso è un **blocco mancante** (missing block). I blocchi mancanti non procurano un danno reale, ma sprecano spazio riducendo la capacità del disco; la soluzione per questo tipo di blocchi è ovvia: il controllore del file system li aggiunge alla lista libera.

Un'altra situazione che si può verificare è in Figura 6.26(c): qui si vede un blocco, il numero 4, che compare due volte nella lista libera (i duplicati si possono avere solo se la lista libera è realmente una lista; con una mappa di bit è impossibile). Ancora una volta la soluzione è semplice: ricostruire la lista libera.

La cosa peggiore che può succedere è che lo stesso blocco dati sia presente in due o più file, come mostrato in Figura 6.26(d) con il blocco 5. Se uno di questi file fosse rimosso, il blocco 5 sarebbe messo nella lista libera, portando a una situazione in cui lo stesso blocco è contemporaneamente in uso e libero allo stesso tempo; se entrambi i file sono rimossi, il blocco sarà messo nella lista libera due volte.

L'azione di recupero che il controllore del file system deve eseguire è quella di allocare un blocco libero, copiarvi il contenuto del blocco 5, e inserire la copia in uno dei file; in questo modo non si modifica il contenuto dell'informazione dei file (sebbene quasi sicuramente uno dei due sarà alterato), ma almeno la struttura del file system è resa coerente. Si dovrebbe segnalare l'errore per permettere all'utente di esaminare il danno.

Oltre al controllo per vedere che le rilevazioni relative ad ogni blocco siano coerenti, il controllore del file system verifica anche la coerenza del sistema di directory: anche questo usa una tabella di contatori, che però sono usati per ogni file piuttosto che per ogni blocco. Esso parte dalla directory radice e ricorsivamente scende lungo l'albero ispezionando ogni directory del file system; per ogni file in ogni directory incrementa il contatore che tiene traccia del numero di usi di quel file. Si ricordi che a causa dei link fisici, un file può essere presente in due o più directory, mentre i link simbolici non contano e non provocano l'incremento del contatore del file target.

Al termine di questo procedimento si ha una lista, indicizzata dal numero di i-node, che dice quante directory contengono ogni file; a questo punto si paragonano questi numeri con i contatori di link memorizzati negli i-node stessi; questi contatori sono inizializzati a 1 e vengono incrementati ogni volta che viene fatto un link (fisico) al file in questione: in un file system coerente, i due contatori avranno lo stesso valore. Tuttavia, si possono avere due tipi di errore: il contatore di link nell'i-node può avere un valore troppo grande o troppo piccolo.

Se il contatore dei link ha un valore maggiore del numero di elementi di directory, allora anche se tutti i file fossero rimossi dalle directory il contatore sarebbe diverso da zero e l'i-node non sarebbe rimosso. Questo errore non è grave, ma spreca spazio sul disco con i file che non sono in alcuna directory; esso dovrebbe essere risistemato assegnando il valore corretto al contatore dei link nell'i-node.

L'altro errore è potenzialmente catastrofico: se due elementi di directory sono collegati a un file, ma l'i-node dice che ve ne è uno solo, quando si rimuove un elemento di directory, il contatore di i-node andrà a zero, e quando un contatore di i-node va a zero, il file system lo contrassegna come inutilizzato e rilascia tutti i suoi blocchi. Questa azione farà sì che una delle directory pointerà adesso a un i-node inutilizzato, i cui blocchi si possono subito assegnare ad altri file: ancora una volta la soluzione consiste nell'imporre come valore del contatore di link nell'i-node il numero effettivo di elementi di directory.

Queste due operazioni, controllo blocchi e directory, sono spesso integrate per ragioni di efficienza (per esempio si effettua una sola passata sull'i-node); sono anche possibili altri controlli euristici. Per esempio, le directory hanno un formato predefinito con i numeri di i-node e i nomi ASCII; se un numero di i-node è più grande del numero di i-node sul disco, la directory è stata danneggiata.

Inoltre, ogni i-node ha una modalità, talvolta legale anche se strana, come 0007, che non permette l'accesso al proprietario e al suo gruppo ma permette agli estranei di leggere, scrivere e eseguire il file; può essere utile almeno evidenziare i file che danno più diritti agli esterni che ai proprietari. Anche le directory con più di 1000 elementi sono sospette; i file allocati nelle directory utenti, ma che sono proprietà del super-user e hanno il bit SETUID abilitato, presentano potenziali problemi di sicurezza, perché quando vanno in esecuzione hanno i diritti del super-user, chiunque sia il proprietario. Con un piccolo sforzo, uno può mettere insieme una lista abbastanza lunga di situazioni legali, ma particolari, che varrebbe la pena segnalare.

Nel capitolo precedente si è discusso il problema del proteggere l'utente da eventuali crash. Qualche file system si preoccupa anche di proteggere l'utente da se stesso. Se l'utente vuole digitare

`rm *.*`

per rimuovere tutti i file che terminano con .o (file oggetto generati dal compilatore), ma accidentalmente digita

`rm * .o`

(si noti lo spazio dopo l'asterisco), `rm` rimuoverà tutti i file nella directory corrente e poi si lamentera che non può trovare il file .o. In MS-DOS e in qualche altro sistema, quando si rimuove un file, si imposta un bit nella directory o nell'i-node per marcare il file come rimosso, ma nessun blocco del disco è rimesso nella lista libera finché non è veramente necessario; perciò, se l'utente scopre l'errore immediatamente, è possibile eseguire un programma di servizio speciale che "ripristina" i file rimossi. In Windows, i file rimossi sono messi nel cestino dei rifiuti, dal quale possono essere recuperati se necessario; natu-

ralmente, non c'è alcun recupero di spazio finché essi non vengono effettivamente cancellati da questa directory.

### 6.3.7 Prestazioni del file system

L'accesso al disco è molto più lento di quello alla memoria: leggere una parola di memoria può richiedere 10 ns, mentre le letture da un disco fisso procedono a 10 MB/s, ossia sono quaranta volte più lente della lettura di una parola di 32 bit; inoltre a questo bisogna aggiungere 5-10 ms di tempo di posizionamento sulla traccia, e inoltre il tempo necessario perché il settore giusto arrivi sotto la testina. Nel caso di lettura di una sola parola, l'accesso in memoria è nell'ordine di un milione di volte più veloce dell'accesso al disco; come risultato di questa differenza nel tempo di accesso, molti file system sono stati progettati con varie ottimizzazioni che permettono di migliorarne le prestazioni. In questa sezione analizzeremo tre di esse.

#### Caching

La tecnica usata più comunemente per ridurre gli accessi al disco è il **block cache** o il **buffer cache** (cache si pronuncia "casc", con la "sc" come in sci, e deriva dal francese *cacher*, che significa nascondere). In questo contesto cache è un insieme di blocchi che logicamente appartengono al disco, ma sono mantenuti in memoria per ragioni legate al miglioramento delle prestazioni.

Si possono usare vari algoritmi per gestire la cache, ma comune mente si controllano tutte le richieste di lettura per vedere se il blocco necessario è nella cache; in questo caso la richiesta di lettura può essere soddisfatta senza accedere al disco. Se il blocco non è in cache, esso è dapprima letto nella cache, e poi copiato ovunque esso sia necessario; richieste successive per lo stesso blocco potranno essere soddisfatte dalla cache.

L'operazione di cache è illustrata in Figura 6.27; siccome ci sono molti blocchi all'interno della cache (spesso migliaia), è necessario determinare velocemente se un particolare blocco si trova nella cache. Un metodo tipico è quello di rappresentare dispositivo e indirizzi del disco in una tabella di hash e cercare il risultato all'interno della tabella; tutti i blocchi con lo stesso valore di hash sono contenuti in un'unica lista collegata, in modo da poter seguire la lista delle collisioni.

Quando si deve caricare un blocco in una cache piena, si rimuove qualche blocco,

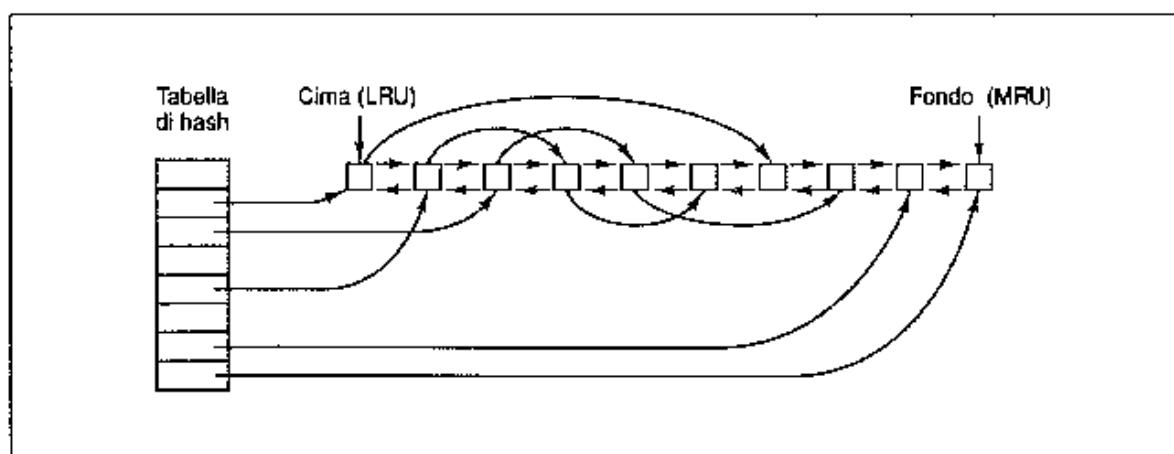


Figura 6.27 Strutture dati del buffer cache.

riscrivendolo sul disco se è stato modificato da quando vi era stato messo. Questa situazione è molto simile alla paginazione, e si possono applicare tutti i tipici algoritmi di paginazione descritti nel Capitolo 3, ad esempio quello FIFO (First In First Out), quello detto "seconda possibilità", o quello LRU (Least Recently Used). Una piacevole differenza tra paginazione e caching è che i riferimenti cache sono relativamente poco frequenti, per cui è possibile tenere tutti i blocchi nell'ordine esatto LRU con liste concatenate.

In Figura 6.27, vediamo che in aggiunta alle catene di collisione che iniziano in corrispondenza della tabella di hash, si ha anche una lista bidirezionale che collega tutti i blocchi nell'ordine di utilizzo, in cima alla quale si trova il blocco usato meno di recente, e in fondo invece si trova il blocco usato più di recente. Quando si fa un riferimento ad un particolare blocco, questo viene spostato dalla sua posizione corrente nella lista bidirezionale e messo in fondo; in questo modo si mantiene l'esatto ordine LRU.

Sfortunatamente vi è un inconveniente: ora che si ha una situazione in cui è applicabile una rigorosa tecnica LRU, viene fuori che LRU non è desiderabile: il problema ha a che fare con i crash e con la consistenza del file system discusse nel paragrafo precedente. Se un blocco critico, come un blocco i-node, è letto nella cache e modificato ma non riscritto sul disco, un crash lascerà il file system in uno stato incoerente; se tale blocco i-node è messo alla fine della catena LRU può passare molto tempo prima che raggiunga l'inizio e sia riscritto sul disco.

Inoltre, alcuni blocchi, come i blocchi i-node, raramente vengono indirizzati due volte in un breve intervallo; queste considerazioni portano a uno schema modificato di LRU, che tiene presente due fattori:

1. Il blocco sarà necessario nell'immediato futuro?
2. Il blocco è essenziale alla consistenza del file system?

Per entrambe le domande, i blocchi si possono dividere in categorie come blocchi i-node, blocchi indiretti, blocchi directory, blocchi completi di dati, e blocchi parzialmente riempiti. I blocchi che probabilmente non saranno di nuovo necessari subito andranno all'inizio piuttosto che in fondo alla lista LRU, così i loro buffer si riuseranno velocemente. I blocchi che possono essere necessari subito, come ad esempio un blocco parzialmente riempito su cui si sta scrivendo, vanno alla fine della lista in modo da essere reperibili per molto tempo.

Il secondo problema non è dipendente dal primo: se il blocco è essenziale per la coerenza del file system (fondamentalmente ogni blocco eccetto i blocchi di dati) ed è stato modificato, dovrà essere scritto sul disco immediatamente, indipendentemente dal punto in cui è sistemato nella lista LRU; scrivendo subito i blocchi critici, si riduce di molto la probabilità che un crash distrugga il file system. Un utente non sarà felice se uno dei suoi file viene rovinato da un crash, ma sarà molto più infelice se va perduto l'intero sistema.

Anche con queste misure per mantenere l'integrità del file system, non è desiderabile tenere i blocchi dati in cache troppo a lungo prima della loro scrittura. Si consideri l'impegno di qualcuno che sta usando un personal computer per scrivere un libro: anche se il nostro scrittore dice periodicamente al programma di scrittura di salvare il file che sta digitando su disco, c'è una buona probabilità che ogni cosa sarà ancora in cache e non sul disco. Se il sistema cade, la struttura del file system non sarà alterata, ma il lavoro di un giorno sarà perduto.

Basta che questa situazione si verifichi qualche volta, neppure molto spesso, per avere un utente profondamente infelice. I sistemi hanno due tipi di approccio per trattare ciò: in UNIX si ha una chiamata di sistema, sync, che forza il trasferimento immediato su disco di tutti i blocchi modificati; quando parte il sistema, un programma, chiamato di solito update, viene fatto partire in background per eseguire un ciclo senza fine che lancia

chiamate di `sync` con una pausa di 30 secondi tra di esse. Come risultato, non si perdonano più di 30 secondi di lavoro per un crash.

In MS-DOS si scrive su disco ogni blocco appena è stato modificato; le cache in cui si riscrivono immediatamente tutti i blocchi modificati sono detti **write-through cache** (letteralmente, scrivi-traverso); essi richiedono molte più operazioni di I/O sul disco delle altre. La differenza tra questi due approcci si può vedere quando un programma riempie tutto un blocco di 1K, un carattere alla volta: UNIX raccoglierà tutti i caratteri nella cache e scriverà il blocco una volta ogni 30 secondi, o comunque quando è rimosso dalla cache; MS-DOS farà un accesso al disco per ogni carattere scritto. Naturalmente molti programmi utilizzano una bufferizzazione interna, per cui normalmente non scrivono un solo carattere, ma una riga o un'unità più grande per ogni chiamata di sistema `write`.

Una conseguenza di questa differenza nella strategia di cache è che la rimozione di un disco (floppy) in un sistema UNIX senza fare una `sync` quasi sempre provocherà perdita di dati, e spesso un file system rovinato, mentre con MS-DOS non sorgono problemi. Queste strategie differenti sono state scelte perché UNIX è stato sviluppato in un ambiente in cui tutti i dischi erano dischi fissi e non rimovibili, mentre MS-DOS nacque nel mondo dei floppy; appena i dischi fissi diventarono comuni anche sui microcomputer, l'approccio UNIX, con la sua maggior efficienza, divenne il modo definitivo di procedere ed oggi è stato scelto anche da Windows per i dischi fissi.

## Lettura anticipata dei blocchi

Una seconda tecnica per migliorare le prestazioni del file system è quella di cercare di portare i blocchi nella cache prima che siano effettivamente necessari, per cercare di migliorare la percentuale di hit (di blocchi trovati nella cache). In particolare, molti file vengono letti in modo sequenziale; quando si chiede ad un file system di produrre il blocco  $k$  di un file, esso esegue la richiesta, e quando ha finito va a dare un'occhiata nella cache per vedere se il blocco  $k + 1$  è già lì. Se non c'è, fa una richiesta di lettura del blocco  $k + 1$ , sperando che quando sarà necessario, sarà già arrivato nella cache, o quanto meno sarà già sulla strada.

Naturalmente, questa strategia di lettura anticipata funziona solo per i file che vengono letti in modo sequenziale; se si accede ad un file in modo casuale, la lettura anticipata non aiuta; in effetti danneggia le prestazioni, occupando la banda del disco in letture di blocchi che non saranno assolutamente necessarie, e rimuovendo blocchi potenzialmente utili dalla cache (ed inoltre occupando ancora più banda nel forzare le scritture su disco dei blocchi eliminati, nel caso siano stati marcati dirty, ossia modificati dall'ultima lettura). Per vedere se sia il caso di utilizzare la strategia di lettura anticipata, il file system può tenere traccia delle sequenze di accesso effettivo ai file aperti; per esempio, può memorizzare in un bit se un file sia acceduto in "modo sequenziale" oppure in "modo casuale". Inizialmente, gli viene dato il beneficio del dubbio impostando il modo sequenziale; però, ogni volta che viene fatta un'operazione di posizionamento, il bit di accesso sequenziale viene cancellato, e nel caso in seguito ricomincia una serie di letture sequenziali, verrà reimpostato. In questo modo, il file system può fare delle ipotesi ragionevoli se sia il caso di effettuare letture anticipate o no; se una volta ogni tanto sbaglia non sarà un disastro, ma semplicemente un po' di banda del disco che andrà sprecata.

## Ridurre il movimento del braccio del disco

La cache e la lettura anticipata non sono i soli modi per aumentare le prestazioni di un file system; un'altra tecnica importante è ridurre il movimento delle testine del disco

sistemando i blocchi a cui si deve accedere con maggior probabilità in sequenza tra di essi, vicini e preferibilmente nello stesso cilindro. Quando si scrive un file, il file system deve allocare i blocchi uno alla volta, man mano che vengono richiesti; se i blocchi liberi sono memorizzati in una mappa di bit, ed essa è interamente in memoria principale, è abbastanza facile scegliere un blocco libero il più vicino possibile al blocco precedente; invece, con una lista libera, una parte della quale è su disco, è molto più difficile allocare insieme i blocchi vicini.

Comunque, anche con una lista libera, si può realizzare un qualche raggruppamento di blocchi. Il trucco consiste nel mantenere traccia della memoria del disco non in blocchi ma in gruppi di blocchi consecutivi. Se i settori sono composti da 512 byte, il sistema potrebbe usare blocchi di 1K (2 settori), ma allocare la memoria del disco in unità di 2 blocchi (4 settori); non è lo stesso che avere un blocco di disco di 2K, poiché la cache potrà usare ancora blocchi di 1K, e i trasferimenti del disco saranno ancora di 1K, ma la lettura sequenziale di un file su un sistema inattivo ridurrà il numero di posizionamenti di un fattore due, migliorando considerevolmente le prestazioni. Una variazione sullo stesso tema è quella di tener presente il posizionamento rotazionale; durante l'allocazione dei blocchi, il sistema tenta di sistemare i blocchi consecutivi di un file nello stesso cilindro.

Un altro collo di bottiglia nelle prestazioni dei sistemi che usano i-node o altre cose equivalenti è che la lettura anche di un file corto richiede due accessi al disco: uno per l'i-node e uno per il blocco. La normale disposizione degli i-node è mostrata in Figura 6.28(a). Qui tutti gli i-node sono vicini all'inizio del disco, così la distanza media tra un i-node e i suoi blocchi sarà circa la metà del numero di cilindri, e richiederà lunghe ricerche.

Un miglioramento delle prestazioni si ottiene sistemando gli i-node a metà del disco, piuttosto che all'inizio, riducendo così di un fattore due la ricerca media tra l'i-node e il primo blocco. Un'altra idea, mostrata in Figura 6.28(b), consiste nel dividere il disco in gruppi di cilindri, ognuno con i suoi propri i-node, blocchi e lista libera (McKusick e altri, 1984). Al momento della creazione di un nuovo file, si può scegliere ogni i-node; una volta scelto, si tenta di trovare un blocco nello stesso gruppo di cilindri dell'i-node, ma se non ce n'è disponibili si usa un blocco in un gruppo di cilindri vicini.

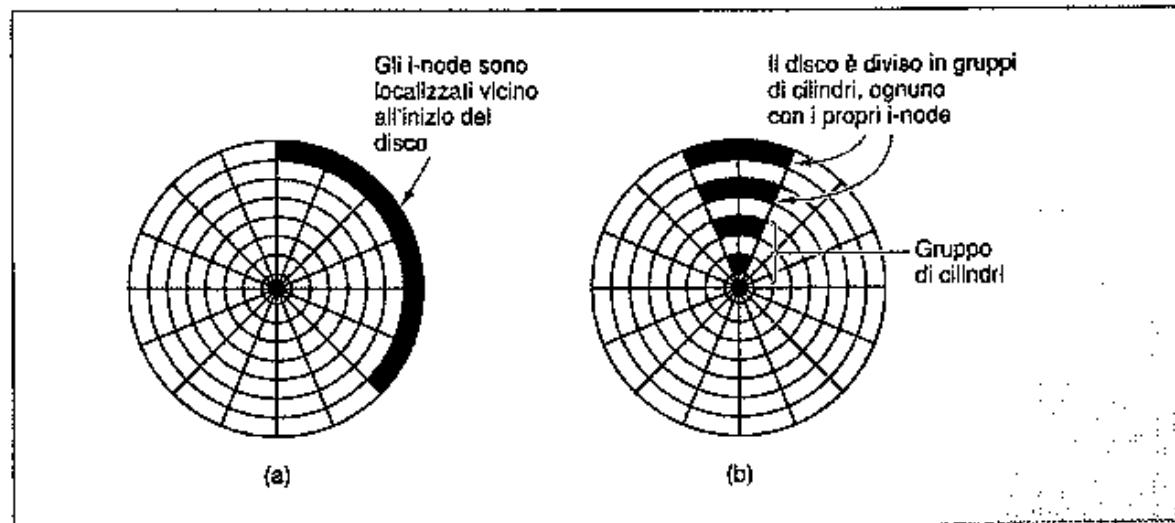


Figura 6.28 [a] i-node posizionati all'inizio del disco. [b] Disco diviso in gruppi di cilindri, ognuno dei quali con i propri blocchi e i-node.

### 6.3.8 File system strutturati a log

I cambiamenti di tecnologia stanno mettendo i file system attuali sotto pressione; in particolare le CPU diventano sempre più veloci, i dischi sempre più grandi ed economici (ma non molto più veloci) e le memorie crescono esponenzialmente di dimensione, mentre il solo parametro che non sta affatto diminuendo di ordini di grandezza è il tempo di posizionamento. La combinazione di questi fattori si traduce in un colpo di bottiglia nelle prestazioni di molti file system. Presso la Berkeley University sono stati portati avanti degli studi volti ad alleviare questo problema progettando un tipo di file system completamente nuovo, chiamato LFS (*file system strutturato a log, Log-structured File System*). In questa sezione descriveremo brevemente come funziona LFS; per una trattazione più completa, si veda (Rosenblum e Ousterhout, 1991).

L'idea guida nella progettazione di LFS è che mentre la CPU diventa più veloce e le memorie RAM sempre più grandi, anche le cache di disco stanno aumentando rapidamente; di conseguenza diventa possibile soddisfare una percentuale cospicua di richieste direttamente dalla cache del file system, senza bisogno di accedere al disco. Segue da questa osservazione che in futuro la maggior parte degli accessi a disco saranno scritte, e che il meccanismo di lettura anticipata usato da alcuni file system per leggere i blocchi prima che essi vengano effettivamente richiesti, non produrrà più considerevoli miglioramenti delle prestazioni.

A peggiorare le cose, nella maggior parte dei file system, le scritture sono fatte per porzioni molto piccole; le scritture piccole sono altamente inefficienti, in quanto una scrittura su disco di 50 µs è normalmente preceduta da un tempo di posizionamento di 10 ms e da 4 ms di ritardo di rotazione; con questi parametri, l'efficienza del disco scende ad un misero 1 per cento.

Per capire da dove vengono le letture piccole, consideriamo la creazione di un nuovo file su di un sistema UNIX; per scrivere questo file, occorre scrivere l'i-node della directory, il blocco della directory, l'i-node del file e il file stesso. Tutte queste scritture possono essere ritardate, ma ciò esporrebbe il file system a seri problemi di consistenza, nel caso avvenga un crash prima del completamento delle scritture; per questa ragione, solitamente le scritture degli i-node vengono fatte immediatamente.

Da questo ragionamento, i progettisti di LFS decisamente di reimplementare il file system di UNIX in modo da raggiungere la piena banda del disco, anche di fronte a un carico di lavoro che si compone principalmente di piccole scritture casuali. L'idea base è quella di strutturare l'intero disco come un log; periodicamente, e quando se ne ha particolarmente bisogno, tutte le scritture pendenti tenute in un buffer in memoria sono raccolte in un unico segmento, e scritte su disco come un unico segmento contiguo alla fine del log. Un singolo segmento, quindi, può contenere i-node, blocchi di directory e blocchi di dati, tutti mescolati insieme; all'inizio di ogni segmento si trova il sommario del segmento, che dice quello che si può trovare all'interno del segmento. Se il segmento medio è di circa 1 MB, si può utilizzare quasi l'intera banda del disco.

In questo tipo di progettazione, gli i-node continuano ad esistere e ad avere la stessa struttura che hanno in UNIX, ma sono distribuiti lungo il log, invece di trovarsi in una posizione prefissata del disco; comunque, quando si localizza un i-node, la localizzazione dei blocchi viene fatta nel solito modo. Ovviamente in questo modo diventa più difficile trovare un i-node, perché il suo indirizzo non può essere semplicemente calcolato a partire dal suo i-numero, come invece accade in UNIX. Per rendere possibile la ricerca degli i-node, si mantiene una mappa degli i-node, indicizzata dall'i-numero: l'ingresso  $i$  della mappa punta all'i-node  $i$  del disco. La mappa viene mantenuta sul disco, ma vengono applicati su di essa meccanismi di cache, quindi le parti di essa maggiormente utilizzate rimangono in memoria per la maggior parte del tempo.

Per riassumere quanto detto finora, tutte le scritture sono inizialmente mantenute in un buffer in memoria, e periodicamente tutte le scritture presenti nel buffer sono scritte su disco in un unico segmento, alla fine del log. Ora, aprire un file significa usare la mappa degli i-node per localizzare l'i-node del file; nel momento in cui l'i-node è localizzato, lo si usa per trovare gli indirizzi dei blocchi del file. Tutti i blocchi si troveranno a loro volta all'interno di segmenti, da qualche parte nel log.

Se i dischi fossero infinitamente larghi, la descrizione appena fatta completerebbe il discorso; però, i dischi reali hanno dimensioni finiti e quindi ad un certo punto il log occuperà l'intero disco, e a quel punto nessun segmento nuovo potrà essere scritto nel log. Fortunatamente, molti segmenti esistenti possono contenere blocchi che non sono più necessari, ad esempio, se un file viene riscritto, il suo i-node punterà a nuovi blocchi, ma quelli vecchi continueranno ad occupare spazio nei segmenti precedentemente scritti.

Per affrontare questo problema, i LFS hanno un thread pulitore (cleaner) che passa il tempo a percorrere ciclicamente il log e a compattarlo; comincia leggendo il sommario del primo segmento del log per vedere quali siano gli i-node e file presenti, poi controlla la mappa degli i-node per vedere se gli i-node sono ancora validi e se i blocchi dei file sono ancora in uso; se non lo sono, vengono eliminati. Gli i-node e il blocchi ancora in uso vanno in memoria per essere scritti nel segmento successivo, mentre il segmento originale viene marcato come libero, quindi può venire usato dal log per i suoi nuovi dati. In questo modo, il pulitore si muove lungo il log, rimuovendo segmenti vecchi dal fondo, e mettendo tutti i dati ancora in vita in memoria per poterli riscrivere nel segmento successivo. Di conseguenza, il disco è un grande buffer circolare, dove il thread di scrittura aggiunge nuovi segmenti in cima e il thread pulitore elimina i vecchi segmenti in fondo.

L'amministrazione, in questo caso, non è semplice, in quanto quando un blocco di un file viene riscritto su un nuovo segmento, l'i-node del file (che sta da qualche parte nel log) deve venire localizzato, aggiornato e messo in memoria per poter essere scritto nel nuovo segmento; la mappa degli i-node deve quindi essere aggiornata in modo da puntare alla nuova copia. Ciononostante, è possibile tenere questo tipo di amministrazione, e i risultati delle prestazioni dimostrano ne vale la pena. Le misurazioni incluse nell'articolo citato in precedenza, mostrano che, in termini di prestazioni, LFS surclassa UNIX di un ordine di grandezza nel caso di piccole scritture, e ha prestazioni paragonabili a quelle di UNIX, o migliori, nel caso di letture o grandi scritture.

## 6.4 Esempi di file system

Nelle sezioni seguenti discuteremo vari esempi di file system, a partire da alcuni piuttosto semplici fino ad altri altamente sofisticati. Siccome i moderni file system UNIX e il file system nativo di Windows 2000 saranno descritti nel capitolo su UNIX (Capitolo 10) e nel capitolo su Windows 2000 (Capitolo 11), non verranno trattati qui, dove invece saranno trattati i loro predecessori.

### 6.4.1 I file system dei CD-ROM

Come nostri primi esempi di file system, consideriamo i file system usati sui CD-ROM; essi sono particolarmente semplici in quanto sono stati progettati per supporti ad una sola scrittura. Tra le altre cose, per esempio, non hanno modo di tenere traccia dei blocchi liberi in quanto i file dei CD-ROM non possono venire cancellati o aggiunti dopo

che il disco è stato prodotto. Nel seguito daremo un'occhiata ai file system per CD-ROM più diffuso e a sue due estensioni.

## Il file system ISO 9660

Lo standard più comune di file system per CD-ROM è stato adottato come standard internazionale nel 1988 sotto il nome di ISO 9660. Quasi ogni CD-ROM presente sul mercato è compatibile con questo standard, in alcuni casi con le estensioni discusse nel seguito. Uno degli scopi di questo standard era quello di fare sì che ogni CD-ROM fosse leggibile da ogni computer, indipendentemente dall'ordinamento dei byte usato, e dal sistema operativo residente. Di conseguenza furono messe alcune limitazioni nel file system, in modo da dare la possibilità, anche al sistema operativo più debole allora in uso (ad esempio MS-DOS) di leggerlo.

I CD-ROM non hanno i cilindri concentrici tipici dei dischi magnetici; hanno invece una spirale continua che contiene i bit in una sequenza lineare (anche se sono possibili posizionamenti lungo la spirale). I bit lungo la spirale sono divisi in blocchi logici (detti anche settori logici) di 2352 byte; alcuni di essi formano i preamboli, i codici di correzione errori e altri overhead, quindi la porzione di carico di ogni blocco logico è di 2048 byte. Quando vengono usati per contenere musica, i CD hanno spazi all'inizio, alla fine, ed intertraccia, che invece non vengono usati per i CD-ROM di dati. Spesso la posizione dei blocchi lungo la spirale è espressa in minuti e secondi, e può essere convertita in numeri di blocco lineari usando il fattore di conversione 1 secondo = 75 blocchi.

ISO 9660 supporta insiemi di CD-ROM con fino a  $2^{16} - 1$  CD in un insieme; il singolo CD-ROM può anche venire partizionato in volumi logici (partizioni). Comunque, nel seguito ci concentreremo su ISO 9660 per CD-ROM singoli e non partizionati.

Ogni CD-ROM comincia con 16 blocchi la cui funzione non è definita dallo standard ISO 9660; un produttore di CD-ROM potrebbe usare questa area per un programma di avvio che permetta ad un computer di poter essere avviato dal CD-ROM, o per qualche altro scopo. Di seguito si trova un blocco contenente il **descrittore di volume primario**, che contiene informazioni generali sul CD-ROM; fra queste informazioni troviamo l'identificatore di sistema (32 byte), l'identificatore di volume (32 byte), l'identificatore del distributore (128 byte) e l'identificatore del preparatore di dati (128 byte). Il produttore può riempire questi campi nel modo desiderato, purché si usino solo lettere maiuscole, numeri ed alcuni elementi di punteggiatura, in modo da assicurare la compatibilità tra piattaforme.

Il descrittore di volume primario contiene anche i nomi di tre file, che possono contenere rispettivamente il riassunto, l'avviso di copyright e informazioni bibliografiche. Inoltre sono presenti numeri chiave, tra cui la dimensione del blocco logico (normalmente 2048, ma in alcuni casi sono anche permessi 4096, 8192 e potenze di 2 superiori), il numero di blocchi del CD-ROM e le date di creazione e scadenza del CD-ROM. Infine, il descrittore di volume primario contiene un elemento di directory per la directory radice, che indica dove questa si può trovare all'interno del CD-ROM (ossia da quale blocco incomincia). A partire da questa directory, è possibile localizzare il resto del file system.

In aggiunta al descrittore di volume primario, un CD-ROM può contenere un descrittore di volume supplementare, che contiene informazioni analoghe al primario, ma che non ci riguarda in questo contesto.

La directory radice e tutte le altre eventuali directory sono composte da un numero variabile di elementi, l'ultimo dei quali contiene un bit che lo marca come ultimo. Anche gli elementi delle stesse directory sono di lunghezza variabile, ognuno è costituito un numero di campi tra 10 e 12, alcuni dei quali sono ASCII, mentre altri sono campi nume-

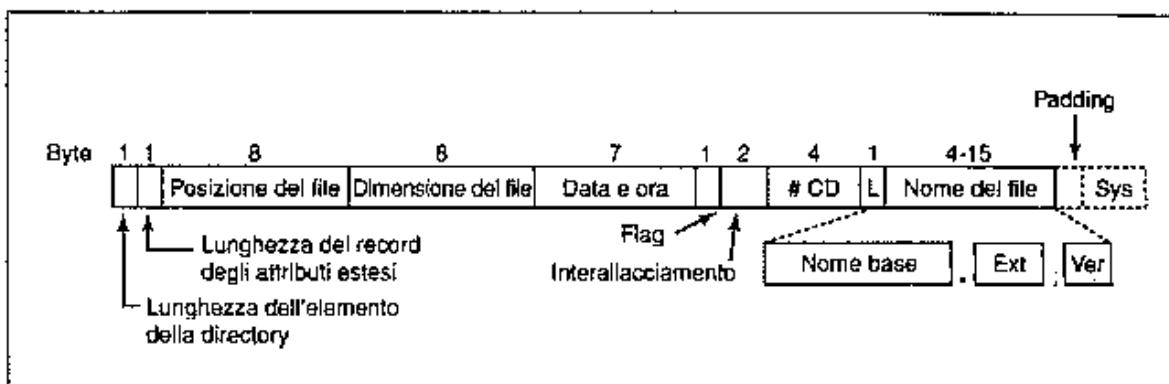


Figura 6.29 Un elemento di directory di ISO 9660.

rici in binario. I campi binari sono codificati due volte, uno in formato little-endian (usato sui Pentium, ad esempio), uno in formato big-endian (usato dalle SPARC, ad esempio); quindi un numero di 16 bit usa 4 byte, e un numero di 32 bit usa 8 byte. L'uso di questa codifica ridondante era necessario per evitare di ferire i sentimenti di qualcuno, quando lo standard fu sviluppato: se lo standard avesse scelto il little-endian, allora le società che sviluppavano prodotti in big-endian si sarebbero sentiti cittadini di seconda categoria, e non avrebbero accettato lo standard. Di conseguenza, il contenuto emotivo di un CD-ROM può essere quantificato e misurato esattamente in Kbyte all'ora di spazio sprecato.

Il formato di un elemento directory di ISO 9660 è illustrato in Figura 6.29. Siccome gli elementi delle directory hanno una lunghezza variabile, il primo campo è un byte che specifica la lunghezza dell'elemento stesso; per definizione, questo campo ha il bit più significativo a sinistra in modo da evitare ambiguità.

Gli elementi di directory possono avere degli attributi estesi; se viene utilizzata questa caratteristica, il secondo byte dell'elemento specifica la lunghezza degli attributi estesi.

Di seguito viene il blocco di inizio del file stesso; i file sono memorizzati come serie contigue di blocchi, così la posizione di un file è completamente specificata dal blocco di inizio e dalla dimensione del file, che è contenuta nel campo successivo.

La data e l'ora in cui il CD-ROM è stato registrato sono memorizzate nel campo successivo, con byte separati per anno, mese, giorno, ora, minuto, secondo e fuso orario. Gli anni iniziano dal 1900, il che significa che i CD-ROM soffriranno del problema Y2156, ossia l'anno successivo a 2155 sarà 1900; questo problema avrebbe potuto essere ritardato scegliendo come anno di inizio il 1988 (l'anno in cui lo standard è stato adottato); se l'avessero fatto, il problema sarebbe stato spostato al 2244, 88 anni in più servono sempre.

Il campo *flag* contiene svariati bit, tra cui un bit che nasconde l'elemento nelle liste (caratteristica copiata da MS-DOS), uno che permette di distinguere un elemento directory da un elemento file, uno che abilita l'uso di attributi estesi e uno che marca l'ultimo elemento di una directory; in questo campo sono anche presenti altri bit, ma non ce ne occuperemo. Il campo successivo riguarda pezzi di file interallacciati in un modo che non viene comunque usato dalle versioni più semplici di ISO 9660, e che quindi non considereremo ulteriormente.

Il campo successivo specifica su quale CD-ROM il file è posizionato; questo perché è possibile che un elemento di directory in un CD-ROM faccia riferimento ad un file che si trova su un altro CD-ROM dell'insieme. In questo modo è possibile costruire una directory master sul primo CD-ROM che lista tutti i file presenti su tutti i CD-ROM dell'insieme completo.

Il campo marcato con *L* in Figura 6.29 fornisce la dimensione del nome del file in byte; esso è seguito dal nome stesso; un nome di file si compone di un nome base, un punto, un'estensione, un punto e virgola e un numero binario di versione (di 1 o 2 byte). Il nome base e l'estensione possono usare lettere maiuscole, le cifre da 0 a 9 e il simbolo di underscore, mentre tutti gli altri caratteri sono proibiti, per fare in modo che tutti i computer possano gestire i nomi dei file. Il nome base può essere lungo fino ad otto caratteri, l'estensione fino a tre caratteri: queste scelte sono state dettate dalla volontà di essere compatibili con MS-DOS. Un nome di file può essere presente in una directory più di una volta, sempre che abbia diversi numeri di versione.

Gli ultimi due campi non sono sempre presenti. Il campo *padding* (imbottitura) viene usato per fare sì che ogni elemento di directory sia costituito da un numero pari di byte, in modo da allineare i campi numerici degli elementi successivi al byte di indirizzo pari; se il padding fosse necessario, si usa un byte con valore 0. Infine, c'è il campo *system use*, la cui funzione e la cui lunghezza non sono specificate, ad eccezione del fatto che deve essere costituito da un numero pari di byte; diversi sistemi lo usano in modo diverso, il Macintosh, ad esempio, tiene qui i flag Finder.

Gli elementi all'interno di una directory sono messi in ordine alfabetico ad eccezione dei primi due: il primo rappresenta la directory stessa, il secondo la directory genitore. In questo senso, questi elementi sono simili agli elementi directory . e .. di UNIX; i file non devono essere nell'ordine della directory.

Non c'è alcun limite esplicito al numero di ingressi in una directory, ma esiste un limite alla profondità della struttura della directory: tale profondità massima dell'albero è otto.

ISO 9660 definisce quelli che sono chiamati tre livelli. Il Livello 1 è il più restrittivo e specifica che i nomi di file devono essere limitati a 8 + 3 caratteri come è stato appena descritto, ed inoltre richiede che i file siano contigui, come è precedentemente stato specificato. Inoltre specifica che i nomi delle directory devono essere limitati ad otto caratteri e non possono avere estensioni. L'uso di questo livello massimizza la probabilità che il CD-ROM possa essere letto da ogni computer.

Il Livello 2 rilassa le restrizioni sulle lunghezze, permettendo che file e directory abbiano nomi lunghi fino a 31 caratteri, ma sempre costituiti dallo stesso insieme di caratteri descritto in precedenza.

Il Livello 3 usa gli stessi limiti sui nomi definiti nel Livello 2, ma rilassa parzialmente i requisiti relativi alla contiguità dei file. Con questo livello, un file può essere formato da diverse sezioni, ognuna delle quali è una serie contigua di blocchi; inoltre la stessa serie può verificarsi più volte all'interno di un file e può addirittura essere presente in due o più file. Se grandi quantità di dati sono ripetute più di una volta, il Livello 3 consente ottimizzazioni dello spazio, dato che non richiede che tutti i dati siano presenti tutte le volte.

## Le estensioni Rock Ridge

Come abbiamo visto, ISO 9660 è restrittivo in molti sensi. Poco tempo dopo la sua nascita, gente proveniente dalla comunità di UNIX cominciò a lavorare su un'estensione che desse la possibilità di rappresentare file system UNIX su un CD-ROM. Queste estensioni vennero chiamate Rock Ridge, dal nome della città nel film di Gene Wilder *Mezzo giorno e mezzo di fuoco*, probabilmente perché ai membri del comitato piaceva quel film.

Le estensioni usano il campo *system use* allo scopo di rendere i CD-ROM Rock Ridge leggibili da ogni computer, tutti gli altri campi mantengono il significato dello standard ISO 9660; ogni sistema che non è consapevole delle estensioni Rock Ridge semplicemente le ignora e vede un normale CD-ROM.

Le estensioni sono divise nei seguenti campi:

1. PX – attributi POSIX.
2. PN – numeri di dispositivo maggiori e minori.
3. SL – link simbolici.
4. NM – nomi alternativi.
5. CL – posizione del figlio.
6. PL – posizione del genitore.
7. RE – rilocazione.
8. TF – timestamp.

Il campo **PX** contiene i bit standard UNIX *rwxrwxrwx* sui permessi del proprietario, del gruppo e degli altri; contiene inoltre altri bit contenuti nel modo parola, quali i bit SETUID e SETGID, e così via.

Per poter rappresentare i dispositivi su un CD-ROM, si usa il campo **PN**, che contiene i numeri maggiori e minori di dispositivo associati al file; in questo modo il contenuto della directory */dev* può essere scritto su un CD-ROM ed in seguito ricostituito correttamente su un altro sistema.

Il campo **SL** si riferisce a link simbolici, e permette ad un file su un file system di riferirsi ad un file di un altro file system.

Probabilmente il campo più importante è **NM**, che permette di associare un secondo nome al file. Questo nome non è soggetto alle restrizioni sulla lunghezza e sul set di caratteri da usare di ISO 9660, rendendo possibile l'espressione di nomi di file arbitrari UNIX su di un CD-ROM.

I tre campi successivi sono usati per superare il limite di ISO 9660, che impone una profondità massima di otto alla struttura delle directory. Usandoli, è possibile specificare che una directory deve essere rilocata, specificando la sua posizione nella gerarchia; si tratta a tutti gli effetti di un modo di aggirare il problema del limite di profondità.

Infine il campo **TF** contiene i tre timestamp inclusi negli i-node di UNIX, ossia il tempo di creazione del file, il tempo in cui è avvenuta l'ultima modifica e il tempo in cui è avvenuto l'ultimo accesso. Insieme, queste estensioni fanno sì che si possa copiare un file system UNIX in un CD-ROM e che lo si possa ripristinare correttamente su un altro sistema.

## Le estensioni Joliet

La comunità UNIX non era la sola a volere estendere ISO 9660; anche la Microsoft trovava lo standard troppo restrittivo (anche se è stato proprio per colpa del MS-DOS della Microsoft che si sono inserite la maggior parte delle restrizioni), quindi ha inventato alcune estensioni che sono state chiamate **Joliet**. Esse sono state progettate per permettere ai file system Windows di essere copiati su un CD-ROM ed in seguito ripristinati, nello stesso modo in cui sono state progettate le estensioni Rock Ridge per UNIX. Quasi tutti i programmi per Windows che usano i CD-ROM supportano Joliet, tra cui tutti i programmi per la masterizzazione di CD registrabili. Solitamente, questi programmi offrono una scelta tra i vari livelli di ISO 9660 e Joliet.

Le principali estensioni fornite da Joliet sono:

1. Nomi di file lunghi.
2. Set di caratteri Unicode.

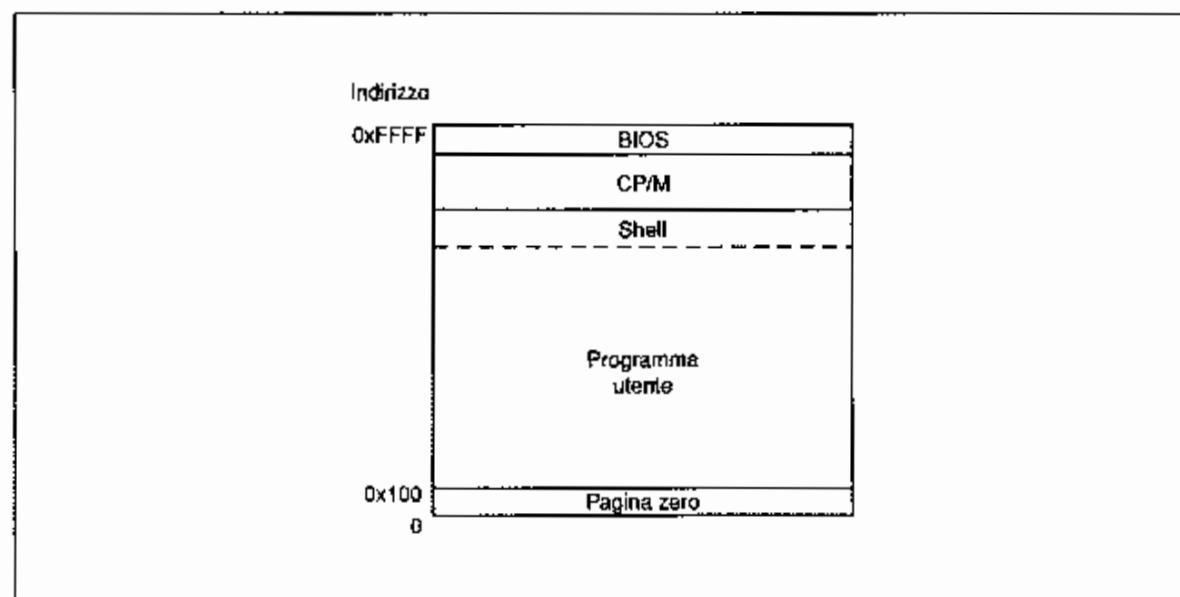
3. Profondità della struttura delle directory maggiore di otto livelli.
4. Nomi di directory con estensioni.

La prima estensione permette che i nomi di file siano lunghi fino a 64 caratteri, la seconda abilita l'uso dell'insieme di caratteri Unicode per i nomi dei file. Questa estensione è importante per il software prodotto per paesi che non usano l'alfabeto latino, quali Giappone, Israele e Grecia. Siccome i caratteri Unicode occupano due byte, la lunghezza massima di un nome di file in Joliet è di 128 caratteri.

Come nel caso di Rock Ridge, le limitazioni nella profondità della struttura delle directory sono state rimosse anche in Joliet; si possono avere strutture tanto profonde quanto si vuole. Infine, i nomi delle directory possono avere estensioni; non è chiaro il perché di questa estensione, in quanto le directory Windows non usano quasi mai estensione, ma forse un giorno lo faranno.

## 6.4.2 Il file system di CP/M

I primi personal computer (allora detti microcomputer) nacquero negli anni ottanta, ed uno dei primi modelli popolari usava il processore a 8 bit Intel 8080, con 4 KB di RAM e un singolo floppy disk di 8 pollici con una capacità di 180 KB. Versioni successive usavano processori leggermente più interessanti (ma sempre a 8 bit) quali Zilog Z80 e avevano fino a 64 KB di RAM e un gigantesco floppy disk di 720 KB come dispositivo di memorizzazione di massa. Nonostante la loro lentezza e la piccola quantità di RAM, quasi tutte queste macchine avevano un sistema operativo basato su disco sorprendentemente potente, chiamato CP/M (*programma di controllo per microcalcolatori, Control Program for Microcomputers*) (Golden e Pechura, 1986). Questo sistema ha dominato la sua era, nello stesso modo in cui MS-DOS e successivamente Windows hanno dominato il mondo dei PC IBM. Due decadi dopo, è svanito senza lasciare traccia (ad eccezione di un piccolo gruppo di ammiratori duri a morire), il che ci fa pensare che sistemi che adesso dominano il mondo potrebbero diventare sconosciuti quando i bambini di oggi diventeranno studenti universitari (Windows chi?).



**Figura 6.30** Struttura della memoria di CP/M.

Vale la pena dare un'occhiata a CP/M per vari motivi; prima di tutto perché è un sistema storicamente molto importante che fu il diretto antenato di MS-DOS; in secondo luogo, perché i progettisti attuali e futuri di sistemi operativi che credono che un computer abbia bisogno di 32 MB soltanto per l'avvio del sistema operativo, potrebbero probabilmente imparare molto dalla semplicità di un sistema che funzionava piuttosto bene con 16 KB di RAM. Terzo, nei decenni futuri i sistemi embedded sono destinati a diventare estremamente diffusi; per motivi di costo, spazio, peso e limiti energetici, i sistemi operativi usati, ad esempio, in orologi, telecamere, radio e cellulari, dovranno essere semplici e scarni, non così diversi da CP/M. Naturalmente, questi sistemi non hanno floppy disk da 8 pollici, ma potranno avere dischi elettronici che usano memoria flash; costruire un file system stile CP/M su un dispositivo di questo tipo è semplice.

La struttura della memoria CP/M è mostrata in Figura 6.30. Al livello più alto della memoria centrale (nella RAM) si trova il BIOS, che contiene una libreria base di 17 chiamate di sistema per l'I/O usate da CP/M (in questa sezione descriveremo CP/M 2.2, che era la versione standard in uso quando CP/M era al massimo della sua popolarità). Queste chiamate leggono e scrivono (da e per) la tastiera, lo schermo e il floppy disk.

Immediatamente sotto al BIOS si trova il sistema operativo vero e proprio; le dimensioni del sistema operativo CP/M 2.2 sono di 3584 byte; incredibile ma vero: un sistema operativo completo in meno di 4 KB! Al di sotto del sistema operativo si trova la shell (l'interprete di comandi di linea), che si mangia altri 2 KB. Il resto delle memoria viene lasciata ai programmi utente, ad eccezione degli ultimi 256 byte, che sono riservati ai vettori di interruzione dell'hardware, ad alcune variabili, e ad un buffer per la linea di comando corrente, che consente ai programmi utente di leggerla.

La ragione della divisione tra BIOS e CP/M (anche se entrambi si trovano nella RAM) è la portabilità: CP/M interagisce con l'hardware solo tramite chiamate al BIOS. Per trasportare CP/M su una nuova macchina, è necessario portare il BIOS e, una volta fatto questo, CP/M può venire installato senza alcuna modifica.

Un sistema CP/M ha solo una directory, che contiene elementi di dimensione fissa (32 byte); la dimensione della directory è fissa, per ogni particolare implementazione, ma può differire da un'implementazione all'altra di CP/M. Tutti i file del sistema sono elencati in questa directory; dopo l'avvio, CP/M legge la directory e calcola una mappa di bit che contiene i blocchi di disco liberi, andando a cercare i blocchi non usati da nessun file. Questa mappa di bit, che è grande solo 23 byte per un disco di 180 KB, viene tenuta in memoria durante l'esecuzione e, quando il sistema viene spento, viene buttata, cioè non viene salvata sul disco. Questo approccio elimina il bisogno di controlli di consistenza sul disco (come *fsck*) e risparmia un blocco del disco (che in percentuale equivale a risparmiare 90 MB su un moderno disco da 16 GB).

Quando un utente scrive un comando, prima di tutto la shell lo copia nel buffer che si trova negli ultimi 256 byte della memoria; poi, cerca il programma da eseguire, lo scrive in memoria all'indirizzo 256 (sopra il vettore delle interruzioni) e trasferisce il controllo dentro di esso, così esso parte, e ricupera i suoi argomenti dal buffer della linea di comando. Il programma ha il permesso di sovrascrivere la shell, nel caso abbia bisogno di spazio. Quando il programma finisce l'esecuzione, genera una chiamata di sistema a CP/M, dicendogli di ricaricale la shell (nel caso fosse stata sovrascritta) ed eseguirla. In breve, questa è più o meno tutta la storia di CP/M.

Oltre a caricare i programmi, CP/M fornisce 38 chiamate di sistema, per la maggior parte servizi per i file, che possono essere usati dai programmi utenti. Tra le più importanti ci sono le chiamate per leggere e scrivere i file: prima di venire letto, un file deve essere aperto: quando CP/M riceve una chiamata di sistema open, deve leggere e cercare all'interno dell'unica directory. La directory non viene tenuta continuamente in memoria, per risparmiare la preziosa RAM; quando CP/M trova l'elemento richiesto, conosce

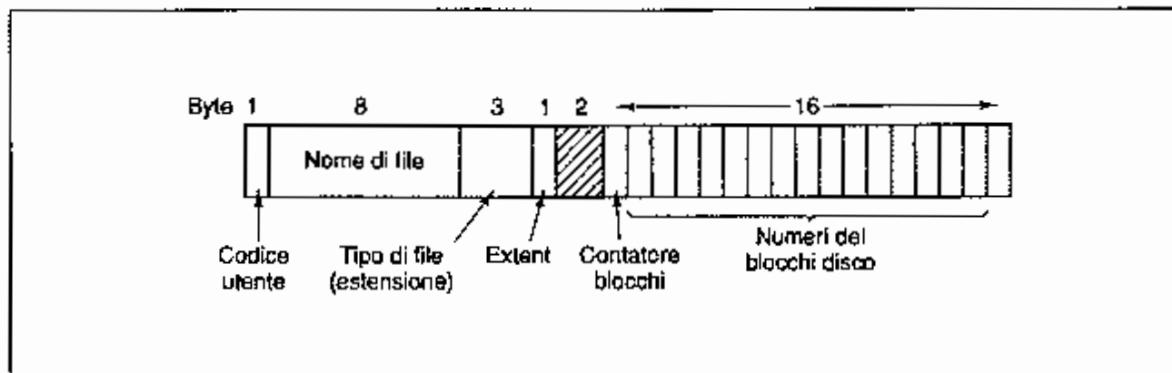


Figura 6.31 Il formato di un elemento di directory in CP/M.

immediatamente i numeri di blocco di disco, perché essi si trovano direttamente all'interno dell'elemento della directory, insieme con tutti gli attributi. Il formato di un elemento di directory è descritto in Figura 6.31.

I campi di Figura 6.31, hanno i seguenti significati: il campo *Codice utente* tiene traccia di quale utente sia proprietario del file; anche se solo una persona alla volta può connettersi ad un sistema CP/M in un dato istante, il sistema supporta più utenti, che possono fare a turno per usare il sistema. Durante la ricerca di un file all'interno della directory, si controllano solo i file appartenenti all'utente collegato al momento. In effetti, ogni utente ha una directory virtuale, senza che ci sia l'overhead di gestire directory multiple.

I due campi successivi forniscono il nome e l'estensione del file; la base può essere lunga fino a 8 caratteri e può essere presente un'estensione facoltativa lunga fino a 3 caratteri. All'interno dei nomi di file sono ammessi soltanto lettere maiuscole, numeri e un sottoinsieme di caratteri speciali; questo modo di scegliere i nomi dei file, 8 + 3 con solo lettere maiuscole, venne adottato in seguito da MS-DOS.

Il campo *Contatore blocchi* indica quanti byte sono contenuti in questo file, misurati in unità di 128 byte (in quanto solitamente le operazioni di I/O coinvolgono settori fisici di 128 byte). L'ultimo blocco da 1 KB può non essere completamente pieno, quindi il sistema non ha modo di conoscere l'esatta dimensione di un file; sta all'utente mettere qualche indicatore di END-OF-FILE, se lo desidera. Gli ultimi 16 campi contendono i numeri dei blocchi di disco; ogni blocco è di 1 KB, quindi la dimensione massima di un file è di 16 KB. Si noti che l'I/O fisico è effettuato in unità di 128 byte (settori) e che quindi si tiene traccia delle dimensioni in settori, ma che i blocchi di file sono allocati in unità da 1 KB (8 settori alla volta) per evitare di rendere l'elemento di directory troppo grande.

Comunque, i progettisti di CP/M si resero conto che alcuni file, anche su un floppy disk di 180 KB, possono diventare più grandi di 16 KB e quindi fu costruita una via di fuga per aggirare il limite dei 16 KB. Un file compreso fra 16 KB e 32 KB deve usare due elementi di directory invece di uno, il primo contenente i primi 16 blocchi, e il secondo contenente i successivi 16; se si superano i 32 KB si usa un terzo elemento di directory, e così via. Il campo *Extent* tiene traccia dell'ordine degli elementi di directory, in modo che il sistema sappia quali 16 KB vengono per primi, quali per secondi, eccetera.

Dopo un'operazione di open, si conoscono tutti gli indirizzi dei blocchi di disco e quindi l'operazione di read è immediata. Anche l'operazione di write è semplice, richiede semplicemente l'allocazione di uno dei blocchi liberi, che può essere scelto tramite la mappa di bit in memoria, e la scrittura del blocco. Blocchi consecutivi di un file non sono messi in blocchi consecutivi sul disco perché il processore 8080 non può elaborare un'interruzione ed iniziare a leggere il blocco successivo in tempo; invece, si usano degli intervalli che permettono di leggere diversi blocchi in una singola rotazione.

CP/M non è certamente l'ultimo grido in termini di file system avanzati, ma è semplice, veloce, e può essere implementato da un programmatore esperto in meno di una settimana. Per molte applicazioni dedicate può costituire il sistema necessario.

### 6.4.3 Il file system di MS-DOS

In prima approssimazione, MS-DOS può essere visto come una versione più grande e migliore di CP/M. Funziona solo su piattaforme Intel, non supporta la multiprogrammazione, funziona solo in modo reale del PC (che originariamente era l'unico modo). La shell ha più caratteristiche, e ci sono più chiamate di sistema, ma la funzione principale del sistema operativo è sempre quella di caricare programmi, gestire tastiera e schermo, e gestire il file system. È a quest'ultima funzionalità che ci interesseremo nel seguito.

Il file system di MS-DOS fu sviluppato molto vicino a quello di CP/M, includendo l'uso dei nomi di file 8 + 3 (maiuscoli). Anche la prima versione (MS-DOS 1.0) era limitata ad un'unica directory, come CP/M; ma a partire da MS-DOS 2.0, le funzionalità del file system furono ampliate notevolmente; il principale miglioramento fu l'aggiunta di un file system gerarchico in cui le directory potevano essere strutturate fino ad una profondità arbitraria. Questo significava che la directory radice (che aveva ancora una dimensione massima fissata) poteva contenere sottodirectory, le quali potevano contenere ulteriori sottodirectory ad infinitum. I link nello stile di UNIX non erano permessi, quindi il file system forniva un albero a partire dalla directory radice.

Tipicamente molti programmi applicativi iniziano creando una sottodirectory nella directory radice, e mettono tutti i loro file lì (o in sottodirectory della directory creata), così che non si creino conflitti tra diverse applicazioni. Siccome le directory stesse sono memorizzate come file, non ci sono limiti al numero di directory o file che possono venire creati. A differenza di CP/M, comunque, non esiste il concetto di utenti diversi in MS-DOS, di conseguenza, l'utente collegato può accedere a tutti i file.

Per leggere un file, un programma MS-DOS deve prima fare una chiamata di sistema open ed ottenere un handle per il file stesso; tale chiamata di sistema specifica un cammino, che può essere assoluto o relativo alla directory di lavoro corrente. Il cammino è analizzato componente per componente fino a localizzare la directory finale e portarla in memoria; ed è qui che si cerca il file da aprire.

Anche se le directory di MS-DOS hanno dimensioni variabili, esse usano, come in CP/M, elementi di directory di 32 byte. Il formato di un elemento di directory in MS-DOS è mostrato in Figura 6.32: contiene il nome di file, gli attributi, il tempo e la data di creazione, il blocco di inizio e la dimensione esatta del file. Nomi di file più corti di 8 + 3 caratteri sono giustificati a sinistra e imbottiti con spazi a destra, in ogni campo separatamente. Il campo Attributi è nuovo e contiene dei bit che indicano se il file è apribile in sola lettura (read-only), se ha bisogno di essere archiviato, se si tratta di un file nascosto o di un file di sistema. I file read-only non sono riscrivibili, il che li protegge da danni accidentali; e il bit archiviato non ha una vera e propria funzione di sistema operativo (ossia, MS-DOS non lo esamina né lo imposta); l'intenzione è quella di permettere a programmi di archiviazione a livello utente di utilizzare questo bit, azzerandolo nel momento in cui il file in questione viene archiviato, e facendo in modo che gli altri programmi lo impostino quando modificano il file; in questo modo, un programma di backup può esaminare questo bit per ogni file e decidere quali file hanno bisogno di backup. Il bit nascosto può essere impostato per impedire che un file appaia quando viene elencato il contenuto di una directory; il suo uso principale è quello di evitare di confondere utenti principianti con file che essi potrebbero non capire. Infine, anche il bit di sistema nasconde i file; in più, i file di sistema non possono venire cancellati acciden-

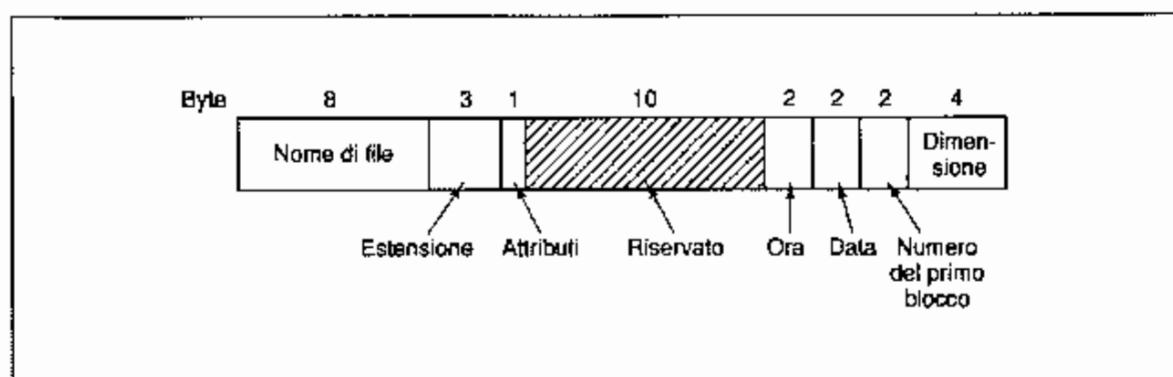


Figura 6.32 Elemento di directory in MS-DOS.

talmente dal comando *del*, e le componenti principali di MS-DOS hanno questo bit impostato.

L'elemento della directory contiene anche la data e l'ora in cui il file fu creato e fu modificato l'ultima volta; l'ora è accurata a  $\pm 2$  s in quanto è immagazzinata in un campo di 2 byte che può memorizzare solo 65536 valori distinti (un giorno contiene invece 86400 secondi distinti). Il campo dell'ora è diviso in secondi (5 bit), minuti (6 bit) e ore (5 bit); la data conta i giorni usando tre sottocampi: giorno (5 bit), mese (4 bit), e anno rispetto al 1980 (7 bit). Con un numero di 7 bit per l'anno, e il computo della data che comincia dal 1980, il massimo anno esprimibile è il 2107. Pertanto MS-DOS ha un problema Y2108 intrinseco: per evitare la catastrofe, gli utenti MS-DOS dovrebbero correre ai ripari il prima possibile. Se MS-DOS avesse rappresentato data e ora in modo combinato come un contatore a 32 bit, avrebbe rappresentato i secondi in modo esatto, e avrebbe rimandato la catastrofe fino al 2116.

A differenza di CP/M, che non memorizza la dimensione esatta dei file, MS-DOS lo fa; siccome per memorizzare la dimensione del file si usa un numero a 32 bit, in teoria i file potrebbero essere grandi fino a 4 GB, però altri limiti (descritti nel seguito) restringono la dimensione massima dei file a 2 GB o meno. Infine, una parte sorprendentemente grande dell'elemento (10 byte) non viene usata.

Un'altra cosa in cui MS-DOS differisce da CP/M è il fatto che non memorizza gli indirizzi di disco del file dentro l'elemento della directory, probabilmente perché i progettisti si accorsero che hard disk grandi (che a quel punto erano comuni sui minicomputer) avrebbero prima o poi raggiunto il mondo di MS-DOS. Invece, MS-DOS tiene traccia dei blocchi dei file tramite una tabella di allocazione dei file tenuta in memoria centrale. L'elemento di directory contiene il numero del primo blocco del file; esso viene usato come indice in una FAT con 64K elementi in memoria centrale; seguendo la catena, si risale a tutti i blocchi di un file. L'operazione della FAT è illustrata in Figura 6.14.

Esistono tre versioni di file system FAT per MS-DOS: FAT-12, FAT-16 e FAT-32, a seconda di quanti bit compongano un indirizzo del disco. In effetti, FAT-32 non è un nome corretto dato che vengono usati solo i 28 bit meno significativi di ogni indirizzo: avrebbe dovuto chiamarsi FAT-28, ma numeri che sono potenze di due suonano molto meglio.

Per tutte le FAT, la dimensione di un blocco del disco può essere un qualche multiplo di 512 byte (eventualmente diverso per ogni partizione), dove l'insieme delle dimensioni di blocco possibili (chiamate **dimensioni dei cluster** dalla Microsoft) differisce da una variante all'altra. La prima versione di MS-DOS usava la FAT-12 con blocchi di 512 byte, il che permetteva una dimensione massima di una partizione di  $2^{12} \times 512$  byte (effettivamente solo  $4086 \times 512$  perché 10 indirizzi di disco venivano usati come indicatori speciali, ad esempio come indicatori di fine del file, blocco danneggiato, eccetera). Con questi

parametri, la dimensione massima di una partizione del disco era di circa 2 MB e la dimensione della FAT in memoria era di 4096 elementi di 2 byte l'uno; usare degli elementi di tabella di 12 bit sarebbe stato troppo lento.

Questo sistema funzionava bene per i floppy disk, ma con l'avvento dei dischi fissi, diventò un problema, che la Microsoft risolse permettendo ulteriori dimensioni dei blocchi possibili, quali 1 KB, 2 KB e 4 KB. Questo cambiamento preservò la struttura e la dimensione della tabella FAT-12, ma permise partizioni di disco fino a 16 MB.

Siccome MS-DOS supportava fino a quattro partizioni per drive di disco, il nuovo file system FAT-16 lavorava con dischi fino a 64 MB; oltre ciò, qualcosa doveva cambiare. Quello che successe fu l'introduzione della FAT-16, con puntatori al disco di 16 bit; in più, si permisero dimensioni di blocco di 8 KB, 16 KB e 32 KB (32768 è la potenza di due più grande che possa essere rappresentata in 16 bit). La tabella FAT-16 occupava in permanenza 128 KB di memoria centrale, ma con le memorie più grandi che a quel punto erano disponibili, venne ampiamente usata e sostituita rapidamente il file system FAT-12. La più grande partizione di disco che può essere supportata dalla FAT-16 è 2 GB (64K elementi di 32 KB l'uno) e il disco più grande 8 GB, ossia quattro partizioni da 2 GB l'una.

Nel caso di lettere commerciali, questo limite non è un problema, ma per memorizzare video digitali usando lo standard DV, un file di 2 GB contiene solo 9 minuti di video. Come conseguenza del fatto che un PC può supportare solo quattro partizioni, il video più lungo che può essere immagazzinato su un disco è di circa 38 minuti, non importa quanto grande sia il disco. Questo limite significa anche che il video più grande che può essere editato on-line è di meno di 19 minuti, dato che sono necessari sul disco sia il file di input che il file di output.

A partire dalla seconda versione di Windows 95, è stato introdotto il nuovo file system FAT-32, con i suoi 28 bit di indirizzamento su disco, e la versione di MS-DOS sottostante Windows 95 fu adattata per supportare la FAT-32. In questo sistema, le partizioni potrebbero in teoria essere di  $2^{28} \times 2^{15}$  byte, ma esse sono effettivamente limitate a 2 TB (2048 GB), in quanto internamente il sistema tiene traccia delle dimensioni delle partizioni in settori da 512 byte, usando numeri da 32 bit e  $2^9 \times 2^{32}$  è 2 TB. La dimensione massima delle partizioni per diverse dimensioni dei blocchi in tutti e tre i modelli di FAT è mostrata in Figura 6.33.

Oltre a supportare dischi più grandi, il file system FAT-32 ha altri due vantaggi rispetto a FAT-16: prima di tutto, un disco da 8 GB che usa una FAT-32 può avere un'unica partizione, mentre usando FAT-16 deve stare su quattro partizioni, che appaiono all'utente

| Dimensione<br>blocco | FAT-12 | FAT-16  | FAT-32 |
|----------------------|--------|---------|--------|
| 0,5 KB               | 2 MB   |         |        |
| 1 KB                 | 4 MB   |         |        |
| 2 KB                 | 8 MB   | 128 MB  |        |
| 4 KB                 | 16 MB  | 256 MB  | 1 TB   |
| 8 KB                 |        | 512 MB  | 2 TB   |
| 16 KB                |        | 1024 MB | 2 TB   |
| 32 KB                |        | 2048 MB | 2 TB   |

Figura 6.33 Dimensione massima delle partizioni per diverse dimensioni dei blocchi. Gli spazi vuoti rappresentano combinazioni vietate.

Windows come i drive logici C:, D:, E:, F:. Sta all'utente decidere quali file mettere in quale drive e tenere traccia di dove stanno le cose.

L'altro vantaggio di FAT-32 rispetto a FAT-16 è che per una data dimensione di una partizione del disco, si può usare una dimensione del blocco più piccola. Per esempio, per una partizione di disco di 2 GB, FAT-16 deve usare blocchi da 32 KB altrimenti, con soli 64K indirizzi, non riesce a coprire l'intera partizione; invece, FAT-32 può usare, ad esempio, blocchi da 4 KB per una partizione di disco da 2 GB. Il vantaggio dei blocchi più piccoli sta nel fatto che molti dei file sono più piccoli di 32 KB e, se la dimensione del blocco è di 32 KB, un file di 10 KB consuma comunque 32 KB. Se il file medio è, ad esempio, di 8 KB, allora con blocchi di 32 KB i tre quarti del disco andranno persi, il che non è un modo terribilmente efficiente di usare il disco. Con un file di 8 KB e un blocco di 4 KB, non c'è spreco di disco, ma il prezzo che si paga è una maggior quantità di memoria RAM occupata dalla FAT: con blocchi da 4 KB e una partizione da 2 GB, ci sono 512K blocchi, quindi la FAT deve avere 512K elementi in memoria (e quindi occupare 2 MB di RAM).

MS-DOS usa la FAT per tenere traccia dei blocchi di disco liberi; ogni blocco che non è usato al momento viene marcato con un codice speciale; quando MS-DOS ha bisogno di un nuovo blocco di disco, cerca nella FAT un elemento contenente questo codice. Quindi non sono necessarie mappe di bit né liste di blocchi liberi.

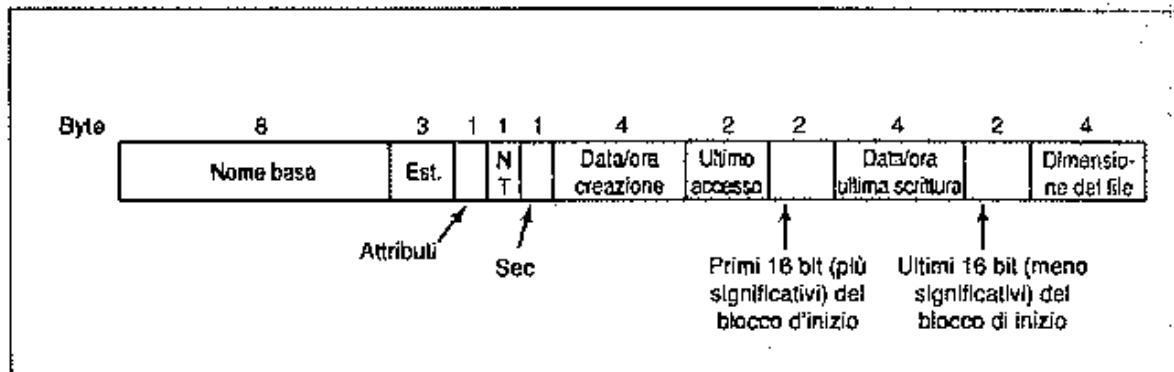
#### 6.4.4 Il file system di Windows 98

La versione originale di Windows 95 usava il file system di MS-DOS, compreso l'uso dei nomi dei file 8 + 3 e i file system FAT-12 e FAT-16. A partire dalla seconda versione di Windows 95, furono permessi nomi di file più lunghi di 8 + 3; inoltre venne introdotta la FAT-32, principalmente allo scopo di permettere partizioni di disco più grandi di 2 GB e dischi più grandi di 8 GB, che a quel punto erano disponibili. Sia i nomi di file lunghi sia la FAT-32 vennero usati in Windows 98, nello stesso modo in cui erano stati usati nella seconda versione di Windows 95. Nel seguito descriviamo queste caratteristiche del file system di Windows 98, che sono state mantenute anche per Windows Me.

Siccome i nomi lunghi sono più eccitanti dal punto di vista utente della struttura della FAT, cominceremo da questi. Un modo per introdurre nomi lunghi avrebbe potuto essere quello di inventare una nuova struttura di directory; il problema con questo approccio è che se la Microsoft lo avesse scelto, le persone che erano ancora nella fase di conversione da Windows 3 a Windows 95 o Windows 98, non avrebbero potuto accedere ai loro file in entrambi i sistemi operativi. All'interno della Microsoft venne presa una decisione politica, cioè che i file creati da Windows 98 avrebbero dovuto essere accessibili anche da Windows 3 (per macchine con doppio avvio). Questo vincolo forzò una progettazione per la gestione di nomi di file lunghi che fosse compatibile con il vecchio sistema di dominazione dei file 8 + 3 di MS-DOS. Siccome questo tipo di vincoli di compatibilità non sono insoliti nella storia dei computer, vale la pena dare un'occhiata in dettaglio a come la Microsoft raggiunse questo scopo.

L'effetto della decisione di essere compatibili con le vecchie versioni significò che la struttura delle directory di Windows 98 doveva essere compatibile con la struttura delle directory di MS-DOS: come abbiamo visto, questa struttura è semplicemente una lista di elementi di 32 byte, come mostrato in Figura 6.32. Questo formato è stato ereditato direttamente da CP/M (che era stato scritto per l'8080), il che dimostra quanto a lungo strutture obsolete possano sopravvivere nel mondo dei computer.

Comunque, a questo punto era possibile allocare i 10 byte non usati nell'elemento mostrato in Figura 6.32, e questo si fece come mostrato in Figura 6.34. Questo cambia-



**Figura 6.34** L'elemento di directory di MS-DOS esteso per Windows 98.

mento non ha nulla a che fare con i nomi di file lunghi, ma viene usato in Windows 98 e quindi vale la pena di capirlo.

I cambiamenti consistono nell'aggiunta di cinque nuovi campi al posto dei 10 byte non utilizzati in precedenza; il campo NT è principalmente presente per compatibilità con Windows NT in termini di visualizzazione dei nomi di file con lettere maiuscole e minuscole nel posto giusto (in MS-DOS tutti i nomi di file erano maiuscoli). Il campo Sec risolve l'impossibilità di memorizzare l'ora in un campo di 16 bit; esso fornisce i bit supplementari perché il nuovo campo *Data/ora di creazione* sia accurato a meno di 10 ms. Un altro campo nuovo è *Ultimo accesso*, che immagazzina la data (ma non l'ora) dell'ultimo accesso al file. Infine, passare alla FAT-32 significa che i numeri dei blocchi sono adesso di 32 bit, quindi serve un altro campo di 16 bit per memorizzare i 16 bit più significativi del numero del blocco di inizio del file.

Adesso arriviamo al cuore del file system di Windows 98: come rappresentare i nomi di file lunghi in modo compatibile all'indietro con MS-DOS. La soluzione scelta era quella di assegnare due nomi a ogni file: un nome di file (potenzialmente) lungo (in Unicode, per compatibilità con Windows NT), e un nome 8 + 3, per compatibilità con MS-DOS; si può accedere ai file usando uno a scelta tra i due nomi. Alla creazione del file, se il suo nome non obbedisce alle regole di MS-DOS (8 + 3 caratteri, nessun carattere Unicode, insieme limitato di caratteri, senza spazi, eccetera), Windows 98 gli inventa un nome MS-DOS utilizzando un certo algoritmo: l'idea principale è quella di prendere i primi sei caratteri del nome, convertirli in lettere maiuscole se necessario, ed aggiungere ~1 per formare il nome base; se questo nome esiste già, allora userà il suffisso ~2 e così via. Inoltre, gli spazi e i punti extra vengono cancellati, e certi caratteri speciali vengono convertiti in "underscore". Ad esempio un file che si chiama *Il tempo è giunto disse il tricheco avrà* il nome compatibile con MS-DOS *ILTEMP~1*. Se successivamente viene creato un file chiamato *Il tempo è giunto disse il coniglio*, ad esso sarà assegnato il nome MS-DOS *ILTEMP~2*, e così via.

Ogni file ha un nome di file compatibile con MS-DOS memorizzato nel formato di directory descritto in Figura 6.34; se un file ha anche un nome lungo, questo sarà memorizzato in uno o più elementi di directory che precedono il nome di file MS-DOS. Ogni elemento relativo al nome lungo contiene fino a 13 caratteri Unicode; gli elementi sono memorizzati in ordine inverso, ossia l'inizio del nome di file si trova immediatamente prima dell'elemento MS-DOS, e i pezzi di nome successivi si trovano prima di esso. Il formato di ogni elemento relativo al nome lungo è dato in Figura 6.35.

Una domanda ovvia è: "Come fa Windows 98 a capire se un elemento di directory contiene un nome di file MS-DOS o (un pezzo di) un nome di file lungo?" La risposta può

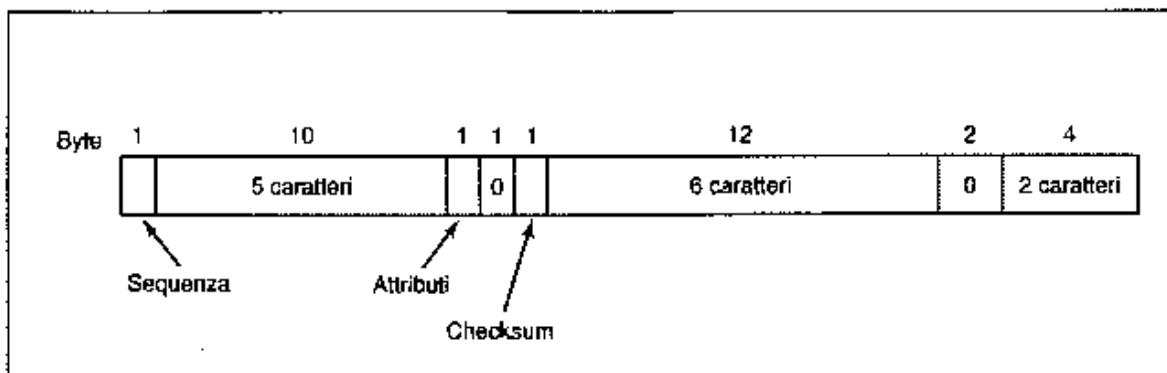


Figura 6.35 Un elemento per [una parte di] un nome di file lungo in Windows 98.

essere trovata nel campo *Attributi*: nel caso di un elemento relativo ad un nome lungo, questo campo ha il valore 0x0F, che rappresenta una combinazione di attributi che altrimenti è impossibile; i vecchi programmi MS-DOS che leggono questo elemento lo ignoreranno considerandolo non valido, ma questo mostra quanto poco ne sanno. I pezzi di nomi sono messi in sequenza usando il primo byte di ogni elemento. L'ultima parte del nome lungo (ossia il primo elemento della sequenza) viene marcata aggiungendo 64 al numero di sequenza. Siccome vengono usati solo 6 bit per il numero della sequenza, il massimo teorico per la lunghezza dei nomi dei file è di  $63 \times 13$  ossia 819 caratteri. In pratica, la lunghezza massima è per ragioni storiche di 260 caratteri.

Ogni elemento relativo ad un nome lungo, contiene un campo *Checksum* (Parità) che serve ad evitare il problema seguente: prima, un programma Windows 98 crea un file con un nome lungo, poi il computer viene avviato in modalità MS-DOS o Windows 3; a questo punto, un vecchio programma rimuove il nome di file MS-DOS dalla directory, ma non rimuove il nome di file lungo che lo precede (questo perché non ne conosce l'esistenza); infine, qualche programma crea un file nuovo che riusa l'elemento di directory appena liberato. A questo punto abbiamo una sequenza valida di elementi relativi ad un nome lungo che precedono un elemento relativo ad un file MS-DOS, che però non ha nulla a che vedere con il file precedente. Il Campo *Checksum* permette a Windows 98 di accorgersi di questa situazione e verificare che il file MS-DOS che segue un nome lungo, in effetti sia relativo allo stesso file. Naturalmente, usando un solo byte, c'è una possibilità su 256 che Windows 98 non si accorga di tale sostituzione.

| Byte | 68 | I | a | n | d | o | A | 0 | C                  |                |       | 0                |                   |
|------|----|---|---|---|---|---|---|---|--------------------|----------------|-------|------------------|-------------------|
|      | 3  | s | o | n | o | t | A | 0 | C                  | r              | o     | t                | t                 |
|      | 2  | n | t | i | n | A | 0 | K | i                  | e              | n     | t                | r                 |
|      | 1  | T | r | e | n | t | A | 0 | C                  | a              | t     | r                | e                 |
|      | T  | R | E | N | T | A | T | S | Tempo di creazione | Ultimo accesso | Primi | Ultima scrittura | Ultimi Dimensione |

Figura 6.36 Un esempio di come un nome lungo viene immagazzinato in Windows 98.

Per avere un esempio di come funzionano i file lunghi, consideriamo l'esempio di Figura 6.36; abbiamo un file dal nome *Trentatre trentini entrarono trotterellando*, che, essendo formato da 43 caratteri, si può certamente definire lungo. Il nome MS-DOS costruito per esso, sarà *TRENTA* e verrà immagazzinato nell'ultimo elemento.

La struttura della directory è ridondante in modo da permettere il rilevamento di problemi, nel caso in cui un vecchio programma Windows 3 abbia fatto confusione all'interno della directory. Il byte relativo al numero di sequenza all'inizio di ogni elemento di directory non è strettamente necessario, ad esempio, in quanto il bit 0x40 marca il primo elemento, ma fornisce ridondanza aggiuntiva. Inoltre, il campo *Ultimi* di Figura 6.36 (la metà inferiore del cluster di inizio) è 0 in tutti gli elementi tranne il primo, anche in questo caso per evitare che vecchi programmi lo fraintendano e rovinino il file system. Il byte *NT* di Figura 6.36 viene usato in NT e ignorato da Windows 98. Il byte *A* contiene gli attributi.

L'implementazione del file system FAT-32 è concettualmente simile all'implementazione del file system FAT-16; però, invece di un array di 65536 elementi, ci sono tanti elementi quanti ne servono a coprire la parte del disco che contiene i dati: se viene usato un milione di blocchi, allora concettualmente la tabella avrà 1 milione di elementi. Per evitare di tenerli tutti insieme in memoria, Windows 98 mantiene una finestra nella tabella, e tiene in memoria solo parti di questa.

#### 6.4.5 Il file system di UNIX V7

Le prime versioni di UNIX avevano già un file system multiutente piuttosto sofisticato, in quanto esso era stato derivato da MULTICS. Nel seguito discuteremo il file system V7, progettato per il PDP-11 e che rese UNIX famoso; esamineremo versioni moderne di UNIX nel Capitolo 10.

Il file system è a forma di albero a partire dalla directory radice, con l'aggiunta di link, che formano un grafo diretto aciclico. I nomi dei file arrivano fino a 14 caratteri e possono contenere ogni carattere ASCII ad eccezione di / (perché è il carattere usato come separatore tra componenti all'interno di un cammino) e NUL (perché viene usato per riempire nomi di file che siano più corti di 14 caratteri). NUL ha come valore numerico 0.

Una directory UNIX contiene un elemento per ogni file contenuto in quella directory; ogni elemento è estremamente semplice in quanto UNIX usa lo schema a i-node illustrato in Figura 6.15. Un elemento di directory contiene solo due campi: il nome del file (14 byte) e il numero dell'i-node per quel file (2 byte), come mostrato in Figura 6.37; questi parametri limitano il numero dei file nel file system a 64K.

Come l'i-node descritto in Figura 6.15, gli i-node di UNIX contengono un certo numero di attributi; essi possono riguardare la dimensione del file, i tre tempi (creazione, ultimo accesso e ultima modifica), proprietario, gruppo proprietario, informazioni relative

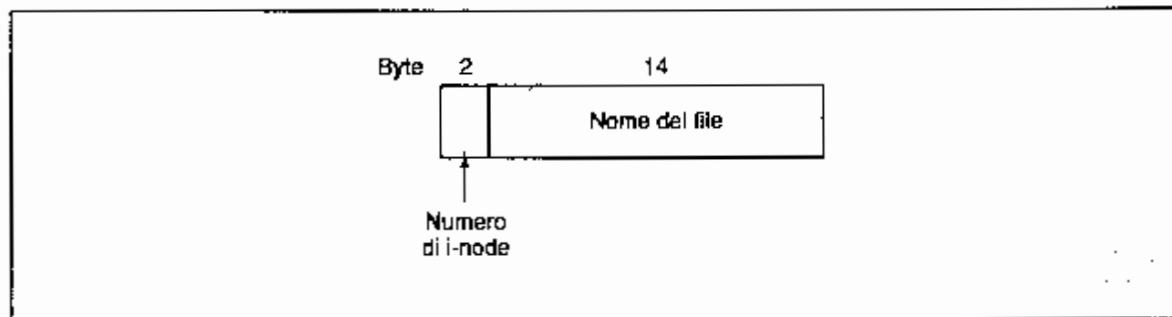


Figura 6.37 Un elemento di directory di UNIX V7.

alla protezione e un contatore del numero di directory che puntano a questo i-node. L'ultimo campo è necessario a causa dei link, ossia quando viene aggiunto un link ad un i-node, il contatore all'interno dell'i-node viene incrementato, mentre quando un link viene rimosso, il contatore è decrementato; e quando arriva a 0, l'i-node viene eliminato e i blocchi di disco relativi vengono inseriti nella lista libera.

Si tiene traccia dei blocchi di disco utilizzando una generalizzazione di quanto mostrato in Figura 6.15, allo scopo di gestire file grandi; i primi 10 indirizzi di disco vengono memorizzati all'interno dell'i-node stesso, quindi per file piccoli, tutta l'informazione necessaria si trova direttamente dentro all'i-node, che viene portato dal disco in memoria centrale all'atto dell'apertura del file. Per file più grandi, uno degli indirizzi contenuti nell'i-node, è l'indirizzo di un blocco di disco chiamato **blocco di indirizzamento singolo indiretto**, che contiene indirizzi aggiuntivi. Se questo non è ancora abbastanza, l'i-node contiene un altro indirizzo, relativo ad un blocco detto **blocco di indirizzamento doppio indiretto**, cioè l'indirizzo di un blocco che contiene un elenco di blocchi di indirizzamento singolo indiretto. Ciascuno di questi blocchi di indirizzamento singolo indiretto punta ad alcune centinaia di blocchi; ma se anche questi non bastassero si può anche usare un **blocco di indirizzamento triplo indiretto**; uno schema completo è dato in Figura 6.38.

Quando un file viene aperto, il file system deve prendere il nome di file fornito e localizzare i suoi blocchi di disco; consideriamo il cammino */usr/ast/mbox*. Useremo UNIX come esempio, ma l'algoritmo è sostanzialmente lo stesso per tutti i sistemi di directory gerarchici: prima di tutto, il file system localizza la directory radice; in UNIX il suo i-node si trova in una posizione fissata del disco. Dal suo i-node, localizza la directory radice, che può essere ovunque nel disco, ma in questo caso supponiamo che si trovi nel blocco 1.

A questo punto legge la directory radice e cerca la prima componente del cammino, *usr*, all'interno della directory radice, allo scopo di procurarsi il suo numero di i-node; una volta noto il numero localizzare l'i-node è semplice, visto che si trovano tutti in posizioni prefissate del disco. Da questo i-node, il file system localizza la posizione della directory */usr* nel disco e, una volta trovata, cerca il numero di i-node della directory successiva, *ast*. Nel momento in cui trova l'elemento relativo ad *ast* e quindi il numero di i-node per la directory */usr/ast*, può trovare la directory stessa e quindi cercare l'elemento relativo a *mbox*; a questo punto, l'i-node relativo a questo file può venire portato in memoria e mantenuto finché il file non viene chiuso. Il processo di ricerca è descritto in Figura 6.39.

I cammini relativi sono analizzati nello stesso modo, ad eccezione che le operazioni di ricerca partono dalla directory di lavoro corrente e non dalla directory radice. Ogni directory ha gli elementi *.* e *..*, che vengono inseriti nel momento in cui una directory viene creata: l'elemento *.* ha il numero di i-node della directory corrente, mentre l'elemento *..* il numero di i-node della directory padre. Pertanto la procedura di cercare *./dick/prog.c*

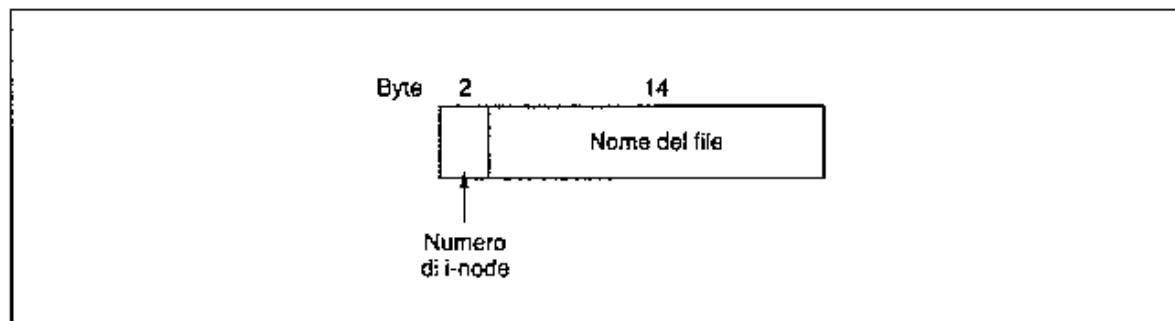


Figura 6.37 Un elemento di directory di UNIX V7.

semplicemente si limita ad una ricerca nella directory di lavoro dell'elemento .. e, una volta trovato il suo numero di i-node, la ricerca prosegue nella directory precedente dove viene cercata la directory *dict*; non sono necessari meccanismi speciali per gestire questi nomi. Dal punto di vista del sistema delle directory, esse sono semplicemente delle normali stringhe scritte in codici ASCII, così come tanti altri nomi.

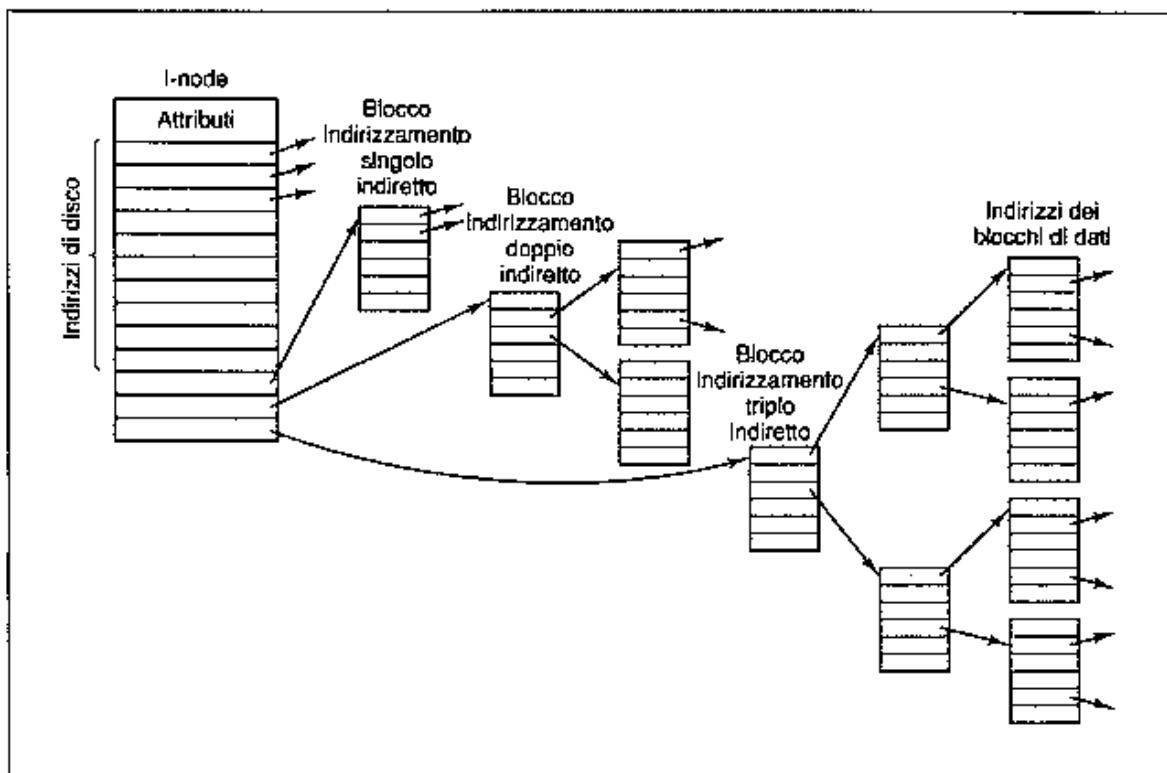


Figura 6.38 Un i-node UNIX.

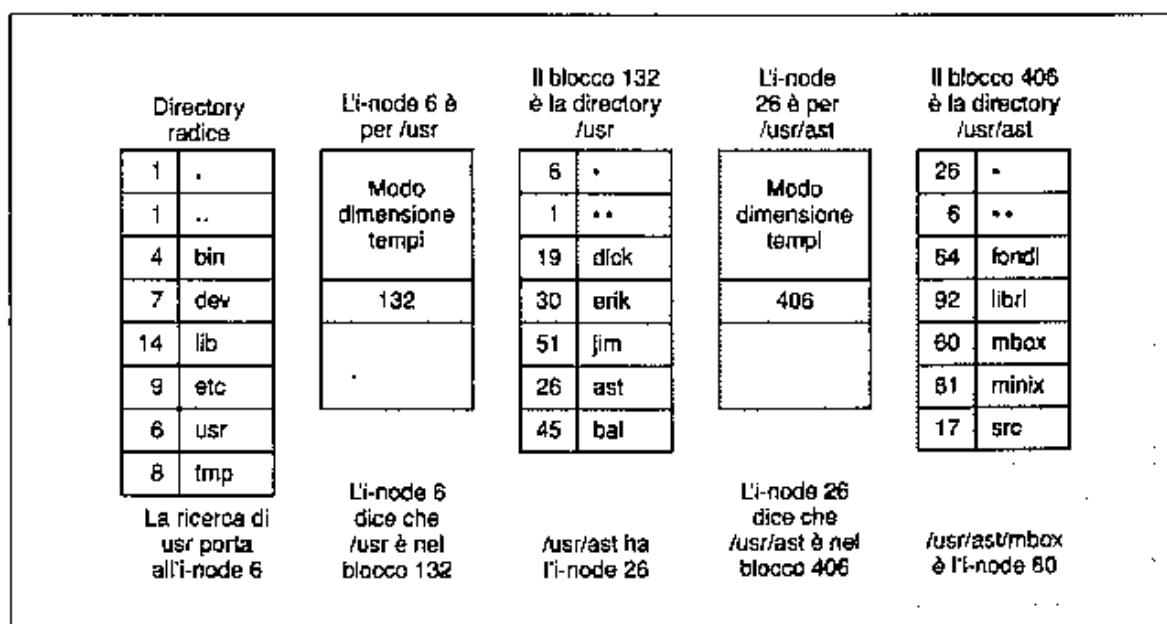


Figura 6.39 Passi relativi alla ricerca di /usr/ast/mbox.

## 6.5 Ricerca sui file system

I file system hanno sempre attirato più ricerca delle altre parti dei sistemi operativi, e questo è vero ancora oggi. Parte di questa ricerca è legata alla struttura dei file system; i file system strutturati a log ed argomenti a essi collegati sono popolari (Matthews et al., 1997; Wang et al., 1999). Il disco logico divide il file system in due livelli distinti: il file system e il disk system (De Jonge et al., 1993). Costruire un file system a partire da livelli impilabili è un altro argomento di ricerca (Heidemann e Popek, 1994).

I file system estensibili sono analoghi ai kernel estensibili che abbiamo visto nel Capitolo 1; essi permettono l'aggiunta di nuove caratteristiche al file system senza doverlo reimplementare completamente da zero (Karpovich et al., 1994; Khalidi e Nelson, 1993).

Un altro argomento di ricerca che ha raggiunto una certa popolarità è quello di fare misurazioni dei contenuti e dell'uso dei file system: sono state misurate le distribuzioni delle dimensioni dei file, la durata della vita dei file, se gli accessi ai file avvengono in modo equo, in che modo operazioni di lettura e scrittura possono essere confrontate e molti altri parametri (Douceur e Bolosky, 1999; Gill et al., 1994; Roselli e Lorch, 2000; Vogels, 1999).

Altri ricercatori hanno analizzato le prestazioni del file system e come migliorarle usando preposizionamenti, meccanismi di cache, meno copiate e altre tecniche; solitamente, questi ricercatori fanno misure, cercano i colli di bottiglia, eliminano almeno uno di essi e controllano che le misure siano migliorate sul nuovo sistema, in modo da validare i loro risultati (Cao et al., 1995; Pai et al., 2000; Patterson et al., 1995).

Un'area a cui molte persone pensano raramente finché non si verifica un disastro è l'area dei backup e dei ripristini dei file system. Anche in questo contesto, sono state pensate nuove idee per migliorare il funzionamento di quanto fatto finora (Chen et al., 1996; Devarakonda et al., 1996; Hutchinson et al., 1999). In qualche modo legata a quest'area è la domanda: cosa fare quando un utente rimuove un file - è meglio rimuoverlo o nasconderlo? Il file system Elephant, ad esempio, non dimentica mai (Santry et al., 1999a; Santry et al., 1999b).

## 6.6 Sommario

Visto dall'esterno, un file system è un insieme di file e directory, con alcune operazioni su di essi: i file si possono leggere e scrivere, le directory si possono creare e distruggere, e i file si possono spostare da directory a directory. La maggior parte dei file system moderni adottano un sistema di directory gerarchiche, in cui le directory possono avere un numero praticamente infinito di sottodirectory.

Visto dall'interno un file system sembra molto diverso: i progettisti di file system si occupano di come allocare lo spazio, e di come il sistema tiene traccia di quali blocchi appartengono a quale file; tra le varie possibilità troviamo i file contigui, le liste collegate, le tabelle di allocazione dei file e gli i-node. Diversi sistemi usano diverse strutture di directory, gli attributi possono trovarsi all'interno delle directory o in altri posti (ad esempio all'interno degli i-node). Lo spazio del disco può essere gestito usando liste libere o mappe di bit. L'affidabilità dei file system può essere migliorata facendo dump incrementali e utilizzando programmi che ripristinino i file system in cattive condizioni. Le prestazioni di un file system sono importanti, e possono essere migliorate in molti modi, con metodi di caching, di letture anticipate, o tramite metodi che posizionano i blocchi di un file l'uno vicino all'altro. File system strutturati a log possono migliorare

le prestazioni raggruppando le operazioni di scrittura in modo da scrivere grandi unità di dati.

Esempi di file system comprendono ISO 9660, CP/M, MS-DOS, Windows 98 e UNIX. Essi si differenziano in molti modi, ad esempio nel modo in cui tengono traccia dei blocchi assegnati ad ogni file, nella struttura delle directory scelta, e nel modo in cui gestiscono lo spazio di disco libero.

## PROBLEMI

1. Dare 5 differenti cammini possibili per il file `/etc/passwd`. Suggerimento: si pensi agli elementi di directory “.” e “..”
2. In Windows, quando un utente dà doppio click su un file elencato da Windows Explorer, viene eseguito un programma che ha questo file come parametro. Elenca due modi possibili con cui il sistema operativo venga a sapere quale programma eseguire.
3. Nei primi sistemi UNIX, i file eseguibili (i file `a.out`) iniziavano con uno specifico magic number (numero magico) e non con uno scelto in modo casuale. Questi file iniziavano con un'intestazione, seguita dal testo e da segmenti di dati. Secondo te, perché veniva scelto un numero specifico per i file eseguibili, mentre tutti gli altri tipi di file avevano un magic number più o meno casuale come prima parola?
4. In Figura 6.4, uno degli attributi è la lunghezza del record. Perché questa grandezza interessa al sistema operativo?
5. La chiamata di sistema `open` di UNIX è assolutamente essenziale? Quali sarebbero le conseguenze di non averla?
6. I sistemi che gestiscono file sequenziali hanno sempre un'operazione per riavvolgere i file. I sistemi che gestiscono file ad accesso casuale hanno anch'essi bisogno di tale operazione?
7. Alcuni sistemi operativi forniscono una chiamata di sistema `rename` per dare a un file un nuovo nome. C'è qualche differenza tra l'usare questa chiamata per rinominare un file e il copiare semplicemente il file su di un nuovo file con il nuovo nome, seguito dalla cancellazione del vecchio?
8. In alcuni sistemi è possibile mappare parte di un file in memoria. Quali restrizioni devono essere imposte dai sistemi? Come si può implementare questo meccanismo di mappatura parziale?
9. Un semplice sistema operativo supporta solo una directory singola ma permette che questa directory abbia un numero arbitrario di file con nomi arbitrariamente lunghi. È possibile simulare un'approssimazione di file system gerarchico? Come?
10. In UNIX e Windows, l'accesso casuale è effettuato utilizzando chiamate di sistema speciali, che muovono il puntatore “posizione corrente” associato al file su di un certo byte del file. Proponi un modo alternativo per effettuare l'accesso casuale che non preveda l'uso di questa chiamata di sistema.
11. Considera l'albero di directory di Figura 6.10. Se la directory di lavoro è `/usr/jim`, qual è il path name assoluto per il file il cui path name relativo è `./fast/x`?
12. L'allocazione contigua del file porta alla frammentazione del disco, come detto nel

testo, in quanto parte dello spazio contenuto nell'ultimo blocco relativo ad un file andrà persa, se la lunghezza del file non è uguale ad un numero intero di blocchi. Si tratta di una frammentazione interna o esterna? Fai un'analogia con quanto discusso nel capitolo precedente.

13. Un modo di usare l'allocazione contigua del disco e non soffrire del problema degli spazi vuoti intermedi è quello di compattare il disco ogni volta che un file viene rimosso. Siccome tutti i file sono contigui, copiare un file richiede un ritardo di posizionamento e rotazione per la lettura del file, seguito da un trasferimento a piena velocità. Scrivere il file nella nuova posizione, richiede lo stesso lavoro. Supponendo che il tempo di posizionamento sia di 5 ms, il ritardo di rotazione di 4 ms, la velocità di trasferimento di 8 MB/s e la dimensione media dei file di 8 KB, quanto tempo occorre per leggere un file dalla memoria centrale e poi riscriverlo su disco nella sua nuova posizione? Usando questi numeri, quanto tempo occorrerebbe per compattare la metà di un disco da 16 GB?
14. Alla luce della risposta data al problema precedente, ha senso utilizzare la compattazione del disco?
15. Alcuni dispositivi digitali richiedono la memorizzazione di dati, ad esempio in forma di file. Nomina un dispositivo moderno che richiede la memorizzazione di file e per il quale l'allocazione contigua sarebbe indicata.
16. Come viene implementato l'accesso casuale ai dati da MS-DOS?
17. Considera l'i-node mostrato in Figura 6.15. Se contiene 10 indirizzi diretti di 4 byte ciascuno, e se tutti i blocchi di disco sono di 1024 KB, qual è il file più grande possibile?
18. Si è suggerito che si potrebbe migliorare l'efficienza e il risparmio di spazio disco se i dati dei file piccoli fossero immagazzinati all'interno dell'i-node. Considerando l'i-node di Figura 6.15, quanti byte di dati potrebbero essere immagazzinati dentro all'i-node?
19. Due studenti di informatica, Carlo e Sandro, stanno discutendo a proposito degli i-node. Carlo afferma che le memorie sono diventate così grandi e così economiche che quando si apre un file, è più semplice e veloce mettere una nuova copia dell'i-node all'interno della tabella degli i-node, invece di cercare all'interno della tabella nell'eventualità che tale i-node sia già presente. Sandro non è d'accordo. Chi ha ragione?
20. Indica un vantaggio dei link fisici rispetto a quelli simbolici, e un vantaggio dei link simbolici rispetto a quelli fisici.
21. Si può tener traccia dello spazio libero su disco con l'utilizzo di una lista libera o una mappa di bit. Se gli indirizzi del disco richiedono  $D$  bit, per un disco con  $B$  blocchi, di cui  $F$  sono liberi, stabilire la condizione al di sotto della quale la lista libera usa meno spazio della mappa di bit. Se il valore di  $D$  è 16 bit, dare una risposta in termini del valore percentuale dello spazio del disco che deve essere libero.
22. L'inizio della mappa di bit dello spazio libero, quando una partizione di disco è stata appena formattata, ha il seguente aspetto: 1000 0000 0000 0000 (il primo blocco è usato dalla directory radice). Il sistema cerca sempre blocchi liberi a partire dal blocco con numero più piccolo, quindi, dopo la scrittura del file A che usa 6 blocchi, la mappa di bit ha il seguente aspetto: 1111 1110 0000 0000. Mostra la mappa di bit dopo ognuna delle seguenti azioni:

- (a) si scrive il file *B*, che occupa 5 blocchi.  
 (b) si cancella il file *A*.  
 (c) si scrive il file *C*, che occupa 8 blocchi.  
 (d) si cancella il file *B*.
23. Cosa succederebbe se la mappa di bit o la lista libera che contengono tutte le informazioni circa i blocchi liberi del disco andassero perse in seguito ad un crash del sistema? Esiste un modo per ripristinare il sistema da questo disastro, o bisogna dire addio al disco? Si discuta la risposta nel caso di sistemi UNIX e sistemi FAT-16 separatamente.
24. Il lavoro notturno di Gino Gufo presso il centro di elaborazione dell'università è quello di cambiare i nastri usati per i backup che vengono effettuati durante la notte. Mentre aspetta il completamento di ogni nastro, lavora alla stesura della sua tesi, che dimostra che le opere di Shakespeare sono state scritte da visitatori extraterrestri. Egli utilizza per scrivere un programma che si trova sulla macchina di cui si fa il backup, siccome è l'unica che hanno. C'è qualche problema in questa organizzazione?
25. I dump incrementali sono stati discussi in dettaglio nel testo. In Windows è facile dire quando occorre fare il dump di un file perché ogni file ha il bit di archiviazione. Questo bit non è presente in UNIX. Come fanno i programmi di backup di UNIX a sapere di quali file fare il dump?
26. Supponi che il file 21 in Figura 6.24 non sia stato modificato dall'ultimo dump. In quale modo le quattro mappe di bit di Figura 6.25 sarebbero diverse?
27. Si è suggerito che la prima parte di ogni file UNIX sia tenuta nello stesso blocco del disco del suo i-node. Quali sono i vantaggi di questa scelta?
28. Considera Figura 6.26. È possibile che per qualche particolare numero di blocco il contatore in entrambe le liste abbia il valore 2? Come si potrebbe correggere questo problema?
29. Le prestazioni di un file system dipendono in maniera critica dal tasso di cache hit (frazione di blocchi trovati nella cache). Se il sistema impiega 1 ms per soddisfare una richiesta nella cache, ma 40 ms quando è necessaria una lettura da disco, dai una formula che esprima il tempo medio richiesto per soddisfare una richiesta con un tasso di cache hit pari a  $h$ . Disegna questa funzione per valori di  $h$  compresi tra 0 e 1,0.
30. Un floppy disk ha 40 cilindri. Una ricerca richiede 6 ms per lo spostamento tra un cilindro e l'altro. Se non si fa attenzione a mettere i blocchi di un file uno vicino all'altro, due blocchi che sono logicamente consecutivi (cioè uno dopo l'altro nel file) saranno mediamente distanti 13 cilindri circa. Comunque, se il sistema operativo fa un tentativo di raggruppare i blocchi legati tra loro, supponiamo che la distanza intermedia tra i blocchi si possa ridurre a due cilindri. Quanto tempo è necessario per leggere un file di 100 blocchi in entrambi i casi, se la latenza rotazionale è 100 ms e il tempo di trasferimento è 25 ms per blocco?
31. Considera l'idea che sta dietro a Figura 6.20, ma per un disco che ha un tempo medio di posizionamento di 8 ms, una frequenza di rotazione di 15000 giri al minuto e 262144 byte per traccia. Quali sono le frequenze di dati per blocchi di dimensione 1 KB, 2 KB e 4 KB, rispettivamente?
32. Un certo file system usa blocchi di disco di 2 KB, e la mediana della dimensione dei suoi file è di 1 KB. Se tutti i file fossero esattamente di 1 KB quale frazione dello spa-

- zio di disco andrebbe persa? Credi che la perdita in un sistema reale sarebbe più alta o più bassa di così? Spiega le tue risposte.
33. CP/M è stato sviluppato per funzionare con piccoli floppy disk come dispositivi di memorizzazione principale. Supponi che venga trasportato su un moderno computer con un ampio disco fisso. Qual è il più grande disco possibile che potrebbe supportare senza dover cambiare la dimensione degli elementi di directory mostrati in Figura 6.31? I campi all'interno dell'elemento di directory e altri parametri di sistema possono venire cambiati, se necessario. Quali cambiamenti faresti?
  34. La tabella FAT-16 di MS-DOS contiene 64K elementi. Supponi che uno dei bit sia stato utilizzato per qualche altro scopo e che la tabella contenga invece esattamente 32768 elementi. Senza ulteriori cambiamenti, quale sarebbe il file più grande possibile di MS-DOS, in queste condizioni?
  35. I file in MS-DOS devono competere per ottenere dello spazio in memoria all'interno della tabella FAT-16. Se un file usa  $k$  elementi, ossia ci sono  $k$  elementi che non sono disponibili per nessun altro file, quale vincolo questo pone sulla lunghezza totale di tutti i file messi insieme?
  36. Un file system UNIX ha blocchi di 1 KB e indirizzi di disco di 4 byte. Qual è la dimensione massima di un file se gli i-node contengono 10 elementi per l'indirizzamento diretto, e un elemento per l'indirizzamento indiretto singolo, uno per il doppio e uno per il triplo indiretto?
  37. Quante operazioni di disco sono necessarie per cercare l'i-node del file */usr/ast/corsi/Sistop/Appunti.t*? Supponi che l'i-node della directory radice sia in memoria, ma che non sia in memoria nessun altro elemento presente nel cammino. Supponi inoltre che tutte le directory stiano in un blocco di disco.
  38. In molti sistemi UNIX, gli i-node sono tenuti all'inizio del disco. Un'alternativa è quella di allocare un'i-node quanto il file viene creato e mettere l'i-node all'inizio del primo blocco del file. Discuti i pro e i contro di questa alternativa.
  39. Scrivi un programma che rovesci i byte di un file, in modo che quello che era l'ultimo byte adesso sia il primo e il primo byte sia ora l'ultimo. Deve funzionare per un file di lunghezza arbitraria, ma dovrebbe essere ragionevolmente efficiente.
  40. Scrivi un programma che inizi da una certa directory e discenda l'albero dei file da quel punto, immagazzinando le dimensioni di tutti i file che trova. Quando ha fatto questo, dovrebbe stampare un istogramma delle dimensione dei file usando l'ampiezza del bin come parametro (ad esempio, con parametro 1024, i file di dimensione dalla 0 alla 1023 vanno in un bin, quelli dalla 1024 alla 2047 varranno in un altro, eccetera).
  41. Scrivi un programma che percorra tutte le directory di un file system UNIX e trovi e localizzi tutti gli i-node con un contatore dei link fisici a due o più e, per ognuno di questi file, elenchi tutti i nomi dei file che puntano ad esso.
  42. Scrivi una nuova versione del programma UNIX *ls*. Questa versione prende come argomento uno o più nomi di directory e per ogni directory elenca tutti i file che essa contiene, una linea per file. Ogni campo dovrebbe essere formattato in modo ragionevole, a seconda del suo tipo. Elenca solo il primo indirizzo di disco, se necessario.
  43. Scrivi CP/M in C o C++. Dai un'occhiata su Web per cercare informazioni su di esso.

# SISTEMI OPERATIVI MULTIMEDIALI

I film digitali, i video clip e la musica stanno diventando un modo sempre più comune di presentare l'informazione e l'intrattenimento per mezzo di un computer; i file audio e video possono essere immagazzinati su un disco ed eseguiti su richiesta, ma le loro caratteristiche sono molto diverse da quelle dei file di testo tradizionali per i quali sono stati progettati gli odierni file system. Di conseguenza, si ha bisogno di nuovi tipi di file system per gestirli, ed inoltre, l'immagazzinamento e l'esecuzione di audio e video pongono nuove richieste allo scheduler e alle altre parti del sistema operativo. Nelle sezioni che seguono studieremo molti di questi problemi e le loro implicazioni per i sistemi operativi progettati per gestire i multimedia.

Di solito, i film digitali vanno sotto il nome di **multimedia**, che letteralmente significa più di un medium; secondo tale definizione, questo libro è un'opera multimediale, poiché contiene, infatti, due media: il testo e le figure (immagini). In ogni caso, la maggioranza delle persone intende con il termine "multimedia" un documento che contiene due o più media *continui*, cioè media che hanno una dimensione temporale; in questo libro utilizzeremo il termine multimedia in tale accezione.

Un altro termine talvolta ambiguo è "video": in senso tecnico, non è altro che la parte di immagini di un film (contrapposta alla parte sonora); infatti, le videocamere digitali e gli apparecchi televisivi spesso hanno due connettori, uno etichettato "video", ed uno "audio", poiché i segnali sono separati. Comunque, il termine video digitale si riferisce normalmente al prodotto completo di suono e immagini, e d'ora in avanti useremo il termine *film* per riferirci al prodotto completo. Si noti che *film*, in questo senso, non è necessariamente una pellicola di due ore, prodotta da uno studio di Hollywood, ad un costo superiore a quello di un Boeing 747. Un clip di notizie di 30 secondi, scaricato dall'home page della CNN, è ugualmente un film, nel senso della nostra definizione; quando ci riferiremo a film molto brevi, li chiameremo anche *video clip*.

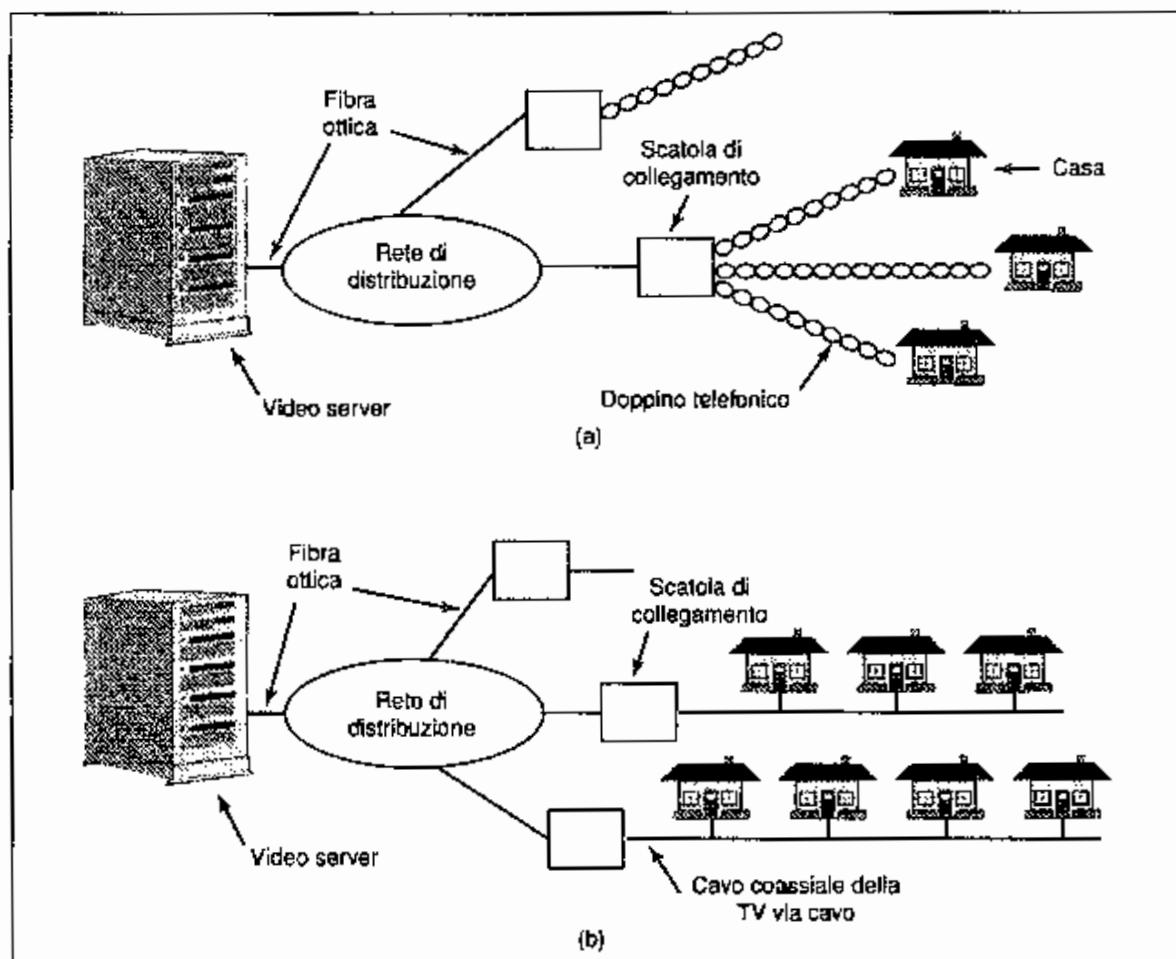
## 7.1 Introduzione ai multimedia

Prima di entrare nella tecnologia dei multimedia, poche parole riguardo alle sue applicazioni presenti e future sono tuttavia utili per inquadrare l'argomento. Su di un singolo computer, multimedia spesso significa guardare (eseguire) un film preregistrato da un **DVD (Digitale Versatile Disk)**. I DVD sono dischi ottici che usano il medesimo supporto di 120 mm di policarbonato (plastica) dei CD-ROM, ma sono registrati ad una maggiore densità, con una capacità fra i 5 e i 17 GB, secondo il formato.

I multimedia sono utilizzati anche per scaricare video clip da Internet, e molte pagine Web consentono di scaricare brevi filmati. A 56 Kbps, scaricare anche un breve video clip richiede parecchio tempo, ma appena si affermeranno tecnologie a più ampia larghezza di banda, come la TV via cavo e l'**ADSL (Asymmetric Digital Subscriber Loop)**, la presenza dei video clip su Internet diventerà massiccia.

Un'altra area nella quale sono supportati i multimedia, è la creazione dei video stessi: esistono sistemi per elaborare i multimedia, e per ottenere migliori prestazioni è necessario che siano eseguiti con sistemi operativi che supportino i multimedia oltre al lavoro tradizionale.

Un altro settore dove i multimedia stanno diventando importanti è quello dei video-giochi, che spesso comprendono brevi video clip per descrivere certi tipi di azioni. I clip sono di solito brevi, ma possono essere numerosi, e quello che serve di volta in volta è



**Figura 7.1** Video su richiesta che utilizza differenti tecnologie di distribuzione locale. (a) ADSL. (b) TV via cavo.

selezionato dinamicamente, secondo le azioni dell'utente. Questi video clip sono sempre più sofisticati.

Infine, il Santo Graal del mondo multimedia è il **video su richiesta** (**video on demand**), in altre parole la possibilità per l'utente, di selezionare da casa un film servendosi del telecomando del proprio televisore (o del mouse), e di visualizzarlo istantaneamente sullo schermo del computer o della TV. Per rendere possibile il video su richiesta è necessaria un'infrastruttura speciale: nella Figura 7.1 vediamo due possibili infrastrutture per il video su richiesta. Ciascuna contiene tre componenti essenziali: uno o più **video server**, una rete di distribuzione, e un dispositivo in ciascuna casa per decodificare il segnale. Il **video server** è un potente computer che memorizza molti film nel suo file system, e li esegue su richiesta. Talvolta, i mainframe sono utilizzati come video server, poiché connettere, ad esempio, mille grandi dischi ad un mainframe è semplice, mentre connettere mille di qualsiasi tipo ad un PC è un problema serio. Gran parte del materiale delle sezioni seguenti riguarda i video server e i loro sistemi operativi.

La rete di distribuzione tra l'utente e il video server deve essere in grado di trasmettere dati ad alta velocità e in tempo reale; la progettazione di tali reti è complessa ed interessante, ma va oltre gli scopi di questo libro. Non diremo nulla in proposito, tranne osservare che queste reti utilizzano sempre fibre ottiche, dal video server sino ai dispositivi di collegamento presso l'abitazione dell'utente. Nei sistemi ADSL, forniti dalle compagnie telefoniche, la linea telefonica normale (doppino) interessa all'incirca solo l'ultimo chilometro di trasmissione, mentre nei sistemi TV via cavo, forniti dai rispettivi operatori, si utilizza, per la distribuzione locale, un cablaggio TV preesistente. ADSL ha il vantaggio di offrire a ciascun utente un canale dedicato, e quindi di garantire la larghezza di banda, che però è bassa (pochi megabit/s), a causa delle limitazioni dell'attuale rete telefonica. La TV via cavo utilizza un cavo coassiale ad ampia larghezza di banda (gigabit/s), condiviso però da diversi utenti, che quindi non dispongono di una larghezza di banda garantita per ciascuno.

L'ultima componente del sistema è il **decodificatore** (decoder), dove termina il cavo ADSL o TV. Si tratta di un normale computer, dotato di diversi chip speciali per la decodifica video e la decompressione; al minimo, contiene CPU, RAM, ROM, e interfaccia ADSL o cavo TV.

Un'alternativa al decoder è l'utilizzo di un normale PC, per visualizzare il film sul monitor; in effetti, la ragione per cui i decoder sono ancora presi in considerazione, benché la maggior parte degli utenti probabilmente possegga già un computer, è che gli operatori del video su richiesta si aspettano che la gente voglia vedere i film nel proprio soggiorno, dove di solito si trova una TV, ma raramente un computer. Da un punto di vista tecnico, utilizzare un PC invece di un decoder ha molto più senso: è più potente, ha un disco capace e uno schermo dotato di una migliore risoluzione. In entrambi i casi, faremo spesso una distinzione tra video server e processo client attivo presso l'utente, che decodifica e visualizza il film. In termini di progettazione del sistema, non è tuttavia molto importante che il processo sia eseguito su un PC oppure su un decoder. Invece, in un sistema per l'elaborazione del video, tutti i processi sono eseguiti sulla stessa macchina, ma si utilizzeranno comunque i termini server e client, per rendere chiaro cosa faccia ogni processo.

Ritornando ai multimedia stessi, ci sono due caratteristiche chiave che occorre comprendere bene per avvalersene con successo:

1. I multimedia utilizzano velocità di trasmissioni dati elevatissima.
2. I multimedia richiedono l'esecuzione in tempo reale.

L'elevata velocità di trasmissione dati deriva dalla natura dell'informazione visiva e acustica. L'occhio e l'orecchio possono elaborare prodigiose quantità di informazione al secondo, e devono essere alimentati ad una velocità tale da produrre un'esperienza vis-

va accettabile. Le velocità di alcune sorgenti digitali multimediali e di alcuni dispositivi hardware sono indicate nella Figura 7.2. Discuteremo alcuni di questi formati di codifica più avanti in questo capitolo; ciò che bisognerebbe notare è l'elevata velocità di trasmissione dati richiesta dai multimedia, la necessità di comprimerli e la quantità di memoria richiesta. Per esempio, un film HDTV di due ore, non compresso, riempie un file di 570 GB. Un video server che memorizza mille film simili, richiede 570 TB di memoria su disco, una quantità non trascurabile per gli standard correnti. È anche degno di nota che, senza compressione dati, l'hardware corrente non potrebbe sostenere la velocità di produzione dei dati. Esamineremo la compressione video più avanti in questo capitolo.

Il secondo problema posto ai sistemi dai multimedia, consiste nella necessità di spedire i dati in tempo reale. La parte video di un film digitale si compone di un certo numero di fotogrammi al secondo. Il sistema NTSC, usato in America e in Giappone, viene eseguito a 30 fotogrammi al secondo (29,97 per i puristi), mentre i sistemi PAL e SECAM, diffusi quasi ovunque nel resto del mondo vengono eseguiti a 25 fotogrammi il secondo (25,00 per i puristi). I fotogrammi devono essere spediti ad intervalli precisi di circa 33,3 ms o 40 ms rispettivamente, altrimenti il film sarebbe saltellante.

Ufficialmente, NTSC significa National Television Standards Committee (Comitato Nazionale per la Standardizzazione Televisiva), ma il modo rozzo con cui il colore è stato definito nello standard al momento della sua comparsa, ha prodotto una barzelletta aziendale: in realtà, significa Never Twice the Same Color (*mai due volte lo stesso colore*). PAL significa Phase Alternative Line (Linee ad alternanza di fase); tecnicamente, è il migliore dei sistemi. SECAM è utilizzato in Francia (s'intendeva proteggere i fabbricanti francesi dalla concorrenza straniera), e significa SEquentiel Couleur Avec Memoire (Colore sequenziale con memoria). SECAM fu adottato anche nell'Europa dell'Est: quando vi fu introdotta la TV, i governi comunisti dell'epoca vollero impedire la visione della TV della Germania Ovest, che trasmetteva in PAL, e scelsero perciò un sistema incompatibile.

L'orecchio è più sensibile dell'occhio, perciò anche una variazione di pochi millisecondi sarà rilevante. La variabilità nella velocità di trasmissione è chiamata jitter e deve essere strettamente limitata per ottenere buone prestazioni. Si noti che lo jitter non è la stessa cosa del ritardo. Se la rete di distribuzione della Fig. 7.1 uniformemente ritarda tutti i bit di esattamente 5000 secondi, il film inizierà un po' dopo, ma sarà visualizzato correttamente. D'altro canto, se ritarda casualmente i fotogrammi tra 100 e 200 ms, il film apparirà come un vecchio film di Chaplin, indipendentemente dal protagonista.

Le proprietà di tempo reale richieste per eseguire in modo accettabile i multimedia sono spesso descritte dai parametri di qualità del servizio. Essi comprendono la larghezza di banda media disponibile, la larghezza di banda di picco, il ritardo minimo e massimo che insieme delimitano lo jitter, e la probabilità di perdita dei bit. Ad esempio,

| Sorgente                       | Mbps  | GB/h | Dispositivo          | Mbps |
|--------------------------------|-------|------|----------------------|------|
| Telefono (PCM)                 | 0,064 | 0,03 | Fast Ethernet        | 100  |
| Musica MP3                     | 0,14  | 0,36 | Disco EIDE           | 133  |
| CD audio                       | 1,4   | 0,62 | Rete ATM OC-3        | 156  |
| Filmati MPEG-2 (640x480)       | 4     | 1,76 | Disco SCSI UltraWide | 320  |
| Videocamera digitale 720 x 480 | 25    | 11   | IEEE 1394 (FireWire) | 400  |
| TV non compressa 640 x 480     | 221   | 97   | Gigabit Ethernet     | 1000 |
| HDTV non compressa 1280 x 720  | 648   | 289  | Disco SCSI Ultra-160 | 1280 |

Figura 7.2 Velocità di trasmissione dati per alcuni multimedia e dispositivi di I/O dalle alte prestazioni. Si noti che 1 Mbps corrisponde a  $10^6$  bit/s, 1 GB a  $2^{30}$  byte.

un operatore di rete potrebbe offrire un servizio garantendo una larghezza di banda media di 4 Mbps, il 99% dei ritardi di trasmissione nell'intervallo fra 105 e 110 ms, e un tasso di perdita dei bit di  $10^{-10}$ , che andrebbe bene per film MPEG-2. L'operatore potrebbe anche offrire un servizio più economico di livello inferiore, con una larghezza di banda media di un Mbps (ad esempio ADSL), nel qual caso dovrebbe esserci un certo compromesso per quanto riguarda la qualità, ad esempio abbassando la risoluzione, diminuendo la velocità di trasmissione dei fotogrammi o eliminando l'informazione relativa al colore, mostrando cioè i film in bianco e nero.

Il modo più comune per fornire garanzie di qualità del servizio consiste nel riservare in anticipo risorse per ciascun nuovo cliente. Le risorse riservate comprendono una parte del tempo di CPU, buffer di memoria, capacità di trasferimento dati dal disco, e larghezza di banda di rete. Se un nuovo cliente si aggiunge e vuole guardare un film, ma il video server o la rete calcola di non avere sufficiente capacità per soddisfarlo, occorre che lo rifiuti, per evitare di degradare il servizio per i clienti già attivi. Di conseguenza, i server multimedia hanno bisogno di schemi di prenotazione delle risorse, e di un algoritmo per il controllo dell'ammissione, per decidere quando possono gestire più lavoro.

## 7.2 File multimediali

Nella maggior parte dei sistemi, un file di testo ordinario si compone di una sequenza lineare di byte senza alcuna struttura che il sistema operativo conosca o di cui si curi; con i multimedia la situazione è più complessa. In primo luogo, video e audio sono completamente differenti: sono captati da dispositivi distinti (chip CCD e microfoni) hanno una diversa struttura interna (il video ha 25-30 fotogrammi/s, l'audio ha 44100 campioni/s), e sono eseguiti da dispositivi diversi (monitor e casse acustiche).

Inoltre, la maggior parte dei film di Hollywood sono ora rivolti ad un pubblico mondiale, la maggior parte del quale non parla inglese. Quest'ultimo problema è affrontato in due modi diversi: per alcuni paesi viene prodotta una traccia sonora aggiuntiva, con le voci doppiate nella lingua locale, ma non gli effetti sonori. In Giappone, tutte le televisioni hanno due canali audio, che permettono allo spettatore di ascoltare i film stranieri in lingua originale oppure doppiati, ed un pulsante sul telecomando permette di selezionare la lingua; in altri paesi si conserva la traccia sonora originale, sottotitolata.

Inoltre, molti film trasmessi in televisione forniscono ora sottotitoli in inglese per permettere ai telespettatori di lingua inglese, ma deboli d'udito, di seguire ugualmente il film. Il risultato è che un film digitale può, di fatto, essere composto di diversi file: un file video, file audio multipli, e file di testo multipli, per i sottotitoli in varie lingue; i DVD possono memorizzare fino a file di sottotitoli in trentadue lingue. La Figura 7.3 illustra un semplice insieme di file multimediali; spiegheremo il significato di avanti veloce e indietro veloce più oltre in questo capitolo.

Di conseguenza, il file system deve tenere traccia di "subfile" multipli per ogni file. Uno schema possibile consiste nel considerare ciascun subfile come un file tradizionale (ad esempio utilizzando un i-node per tenere traccia dei suoi blocchi) e nell'introdurre una nuova struttura dati che elenchi tutti i subfile per ogni file multimediale. Un altro modo consiste nell'introdurre una sorta di i-node bidimensionale le cui colonne elencino i blocchi di ciascun subfile. In generale, l'organizzazione deve essere tale che lo spettatore possa scegliere al momento la traccia audio e i sottotitoli desiderati.

In tutti i casi è anche necessaria una modalità di sincronizzazione dei subfile, in modo che quando la traccia audio selezionata viene eseguita, rimanga in sincronia col video. Se l'audio e il video non sono perfettamente sincronizzati, lo spettatore può udire le parole

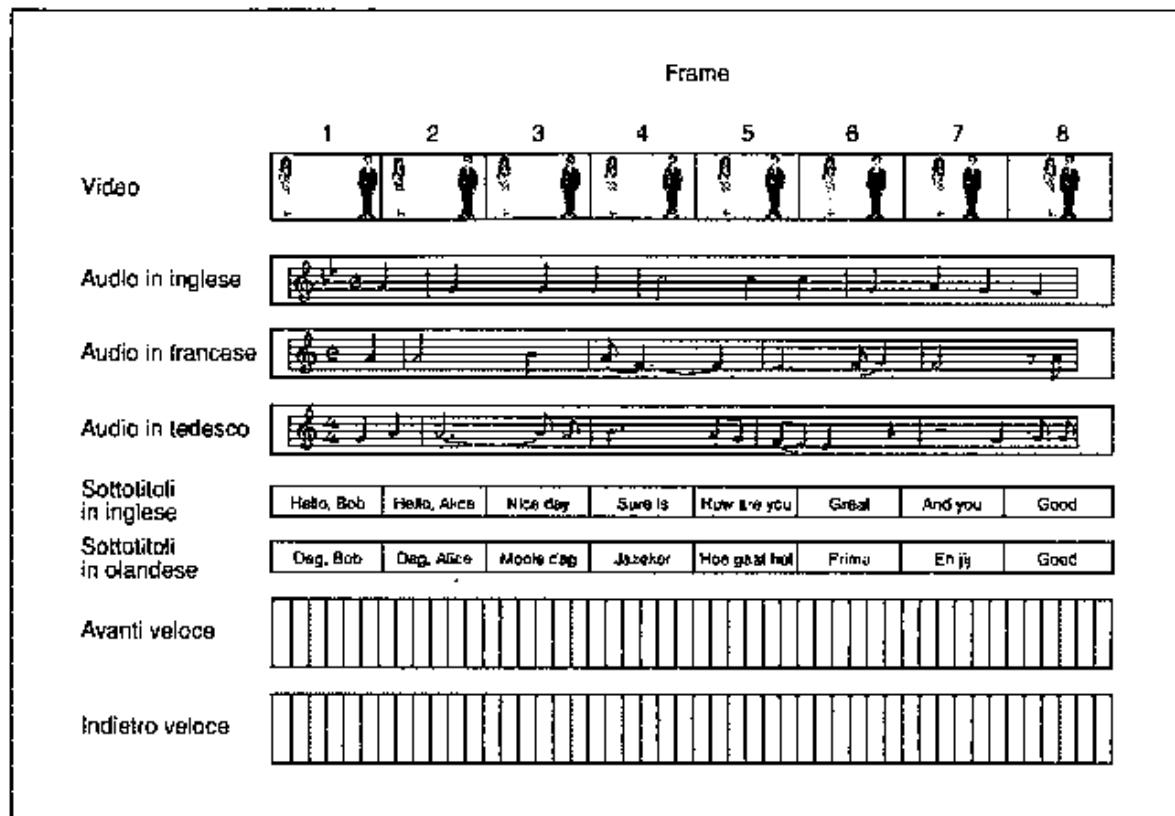


Figura 7.3 Un film può essere composto di diversi file.

di un attore prima o dopo che muova le labbra, il che è facilmente avvertibile e assai fastidioso.

Per capire meglio come siano organizzati i file multimediali, è necessario comprendere piuttosto in dettaglio il funzionamento degli audio e dei video. Introdurremo ora questi argomenti.

## 7.2.1 Codifica audio

Un'onda sonora è un'onda acustica (di pressione) monodimensionale. Quando essa entra nell'orecchio, il timpano vibra, causando la vibrazione delle piccole ossa dell'orecchio interno, che mandano impulsi nervosi al cervello: tali impulsi sono percepiti come suono dall'ascoltatore. In modo simile, quando un'onda acustica colpisce un microfono, esso genera un segnale elettrico, rappresentando l'ampiezza del suono in funzione del tempo.

L'intervallo di percezione delle frequenze da parte dell'orecchio umano varia da 20 Hz a 20000 Hz, benché alcuni animali, in particolare il cane, possano udire frequenze più elevate. L'orecchio percepisce i suoni logaritmicamente, cosicché il rapporto di due suoni con ampiezze *A* e *B* è convenzionalmente espressa in dB (decibel), secondo la formula

$$dB = 20 \log_{10}(A/B)$$

Se si definisce il limite inferiore dell'udibilità (una pressione di circa 0,0003 dyne/cm<sup>2</sup>), per un'onda sonora sinusoidale ad 1 kHz, come 0 dB, una conversazione ordinaria avviene a circa 50 dB e la soglia del dolore è a circa 120 dB, con un range dinamico di circa un milione di volte. Per evitare qualsiasi confusione, i precedenti *A* e *B* sono ampiezze.

Se usassimo la potenza, che è proporzionale al quadrato dell'ampiezza, il coefficiente del logaritmo sarebbe 10 e non 20.

Le onde audio possono essere convertite in formato digitale da un **ADC (Analog Digital Converter, convertitore analogico digitale)**: esso riceve un voltaggio elettrico come input e genera un numero binario come output. Nella Figura 7.4 (a) si vede un esempio di onda sinusoidale; per rappresentare questo segnale in modo digitale, lo si può campionare ogni  $\Delta T$  secondi come mostrato dall'altezza delle barre nella Figura 7.4 (b). Se un'onda sonora non è un'onda sinusoidale pura, ma una sovrapposizione lineare di onde sinusoidali dove la più alta frequenza delle componenti presenti è  $f$ , allora è sufficiente campionare ad una frequenza pari a  $2f$ . Questo risultato è stato provato matematicamente da H. Nyquist nel 1924. Campionare a maggior frequenza non serve, poiché le più alte frequenze che tale campionamento individuare non sono presenti.

I campioni digitali non sono sempre esatti. I campioni della Figura 7.4 (c) possono assumere solo nove valori, da -1,00 a +1,00 a passi di 0,25. Di conseguenza, sono necessari 4 bit per rappresentarli tutti. Un campione a 8-bit permetterebbe 256 valori distinti ed un campione a 16-bit permetterebbe 65536 valori distinti. L'errore introdotto dal numero finito di bit disponibile per ogni campione è chiamato **rumore di quantizzazione**. Se è troppo grande, l'orecchio è in grado di percepirllo.

Due noti esempi di suoni campionati sono il telefono e i CD audio. Il sistema **Pulse code modulation** (modulazione a codice di impulsi) è utilizzato nei sistemi telefonici e impiega campioni di 7 bit (Nord America e Giappone), o 8 bit (Europa), 8000 volte il secondo. Questo sistema permette una velocità di trasmissione di 56000 bps o 64000 bps, ma con solo 8000 campioni/s, le frequenze al di sopra dei 4 kHz sono perdute.

I CD audio hanno frequenze di campionamento di 44100 campioni/s, sufficienti per catturare frequenze fino a 22050 Hz, che è soddisfacente per le persone, ma non per i cani. I campioni sono a 16 bit e sono lineari rispetto all'intervallo delle ampiezze; si noti che tali campioni permettono solo 65536 valori distinti anche se il range dinamico dell'orecchio è circa di un milione, se misurato in passi grandi quanto il più piccolo suono udibile. Quindi, utilizzando solo 16 bit per campione, s'introduce un certo errore di quantizzazione (benché l'intero intervallo dinamico non sia coperto: i CD non devono raggiungere la soglia del dolore). Con 44100 campioni/s di 16 bit ciascuno, un CD audio necessita di una larghezza di banda di 705,6 Kbps per la riproduzione mono, di 1411 Mbps per la riproduzione stereo (Figura 7.2). La compressione audio è possibile e si basa su modelli-

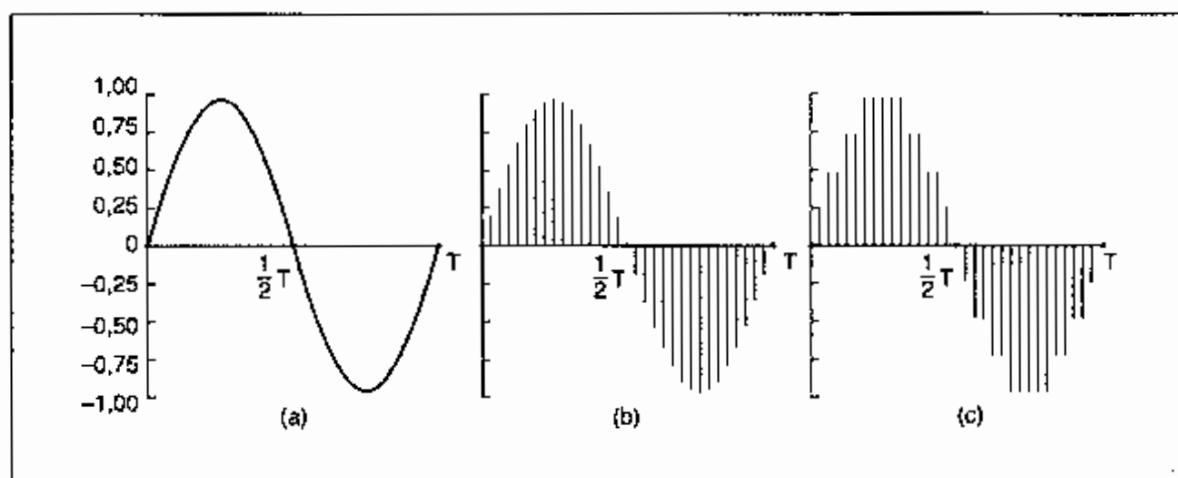


Figura 7.4 [a] Un'onda sinusoidale. [b] Campionamento dell'onda sinusoidale. [c] Quantizzazione dei campioni a 4 bit.

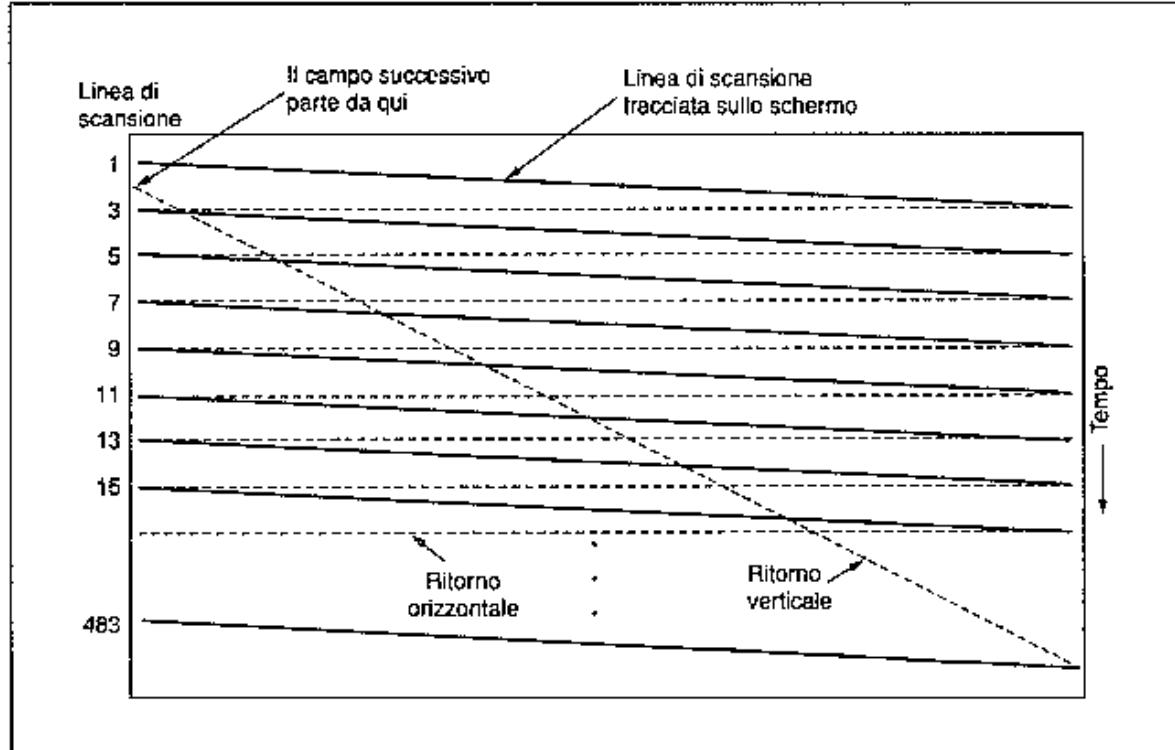
li psicoacustici riferiti al funzionamento dell'udito umano; una compressione di 10x è possibile tramite il sistema MPEG Layer 3 (MP3), i riproduttori portatili di musica per questo formato sono divenuti comuni in anni recenti.

Il suono digitalizzato può essere facilmente elaborato dai computer a livello software. Esistono dozzine di programmi per PC che permettono agli utenti di registrare, visualizzare, elaborare, mixare e memorizzare onde sonore da sorgenti multiple. Praticamente, oggi tutta la registrazione ed elaborazione professionale del suono è digitale.

## 7.2.2 Codifica video

Una proprietà dell'occhio umano è di mantenere per alcuni millisecondi un'immagine proiettata sulla retina prima che essa svanisca. Se una sequenza di immagini è proiettata a cinquanta o più immagini il secondo, l'occhio non si accorge che sta guardando immagini discrete. Tutti i sistemi video e i film sfruttano questo principio per produrre immagini in movimento.

Al fine di comprendere i sistemi video, è meglio partire dal semplice, antiquato televisore in bianco e nero. Per rappresentare l'immagine bidimensionale, a partire da una tensione monodimensionale in funzione del tempo, la telecamera effettua una scansione, per mezzo di un fascio di elettroni, rapidamente da sinistra a destra e lentamente dall'alto in basso, registrando l'intensità luminosa sullo schermo. Alla fine della scansione, chiamata **frame**, il fascio riparte. Quest'intensità in funzione del tempo rappresenta la trasmissione, e i ricevitori ripetono il processo di scansione per ricostruire l'immagine. La Figura 7.5 mostra lo schema di scansione utilizzato sia dalla telecamera sia dal ricevitore. (Per inciso, le telecamere CCD integrano piuttosto che scandire, ma alcune telecamere e tutti i monitor CRT effettuano la scansione).



**Figura 7.5** Lo schema di scansione utilizzato per il video e la televisione NTSC.

I parametri esatti della scansione variano da paese a paese: NTSC ha 525 linee di scansione, un rapporto delle dimensioni orizzontale e verticale di 4:3, e 30 frame/s; i sistemi europei PAL e SECAM hanno 625 linee di scansione, lo stesso rapporto di 4:3 e 25 frame/s. In entrambi i sistemi alcune linee in cima e in fondo non sono visualizzate (per approssimare un'immagine rettangolare sul CRT originale arrotondato): sono visualizzate solo 483 delle 525 linee di scansione NTSC (e 576 delle 625 linee di scansione PAL/SECAM).

Mentre 25 frame/s sono sufficienti per rendere un movimento regolare, a quella frequenza di frame molte persone, specialmente le più anziane, percepiscono uno sfarfallio dell'immagine (poiché la vecchia immagine ha abbandonato la retina prima che la nuova appaia). Piuttosto che incrementare la frequenza dei frame, che richiederebbe di utilizzare larghezze di banda più limitate, si sfrutta un diverso approccio. Invece di visualizzare le linee di scansione in ordine dall'alto in basso, si visualizzano prima tutte le linee di scansione dispari, quindi quelle pari. Ciascuno di questi mezzi frame è chiamato campo. Gli esperimenti hanno mostrato che, per quanto le persone avvertano lo sfarfallio a 25 frame/s, non lo avvertono a 50 campi/s. Questa tecnica è chiamata interallacciamento. La televisione o il video non interallacciato sono detti progressivi.

Il video a colori usa lo stesso modello di scansione del monocromatico (bianco e nero), però, invece di visualizzare l'immagine con un unico fascio, ne utilizza tre, che si muovono all'unisono, uno per ciascuno dei tre colori additivi primari: rosso, verde e blu (RGB). Questa tecnica funziona perché ogni colore può essere costruito per sovrapposizione lineare di rosso, verde e blu, con le opportune intensità. In ogni caso, per la trasmissione su un canale singolo i tre segnali di colore devono essere combinati in un singolo segnale composito.

Per permettere di visualizzare le trasmissioni a colori sui ricevitori in bianco e nero, tutti e tre i sistemi combinano linearmente i segnali RGB in un segnale di luminanza (luminosità) e due segnali di crominanza (colore), benché tutti usino coefficienti diversi per costruire questi segnali da quelli RGB. Abbastanza curiosamente, l'occhio è molto più sensibile al segnale di luminanza che a quelli di crominanza, cosicché questi ultimi non devono essere trasmessi così accuratamente. Di conseguenza, il segnale di luminanza può essere trasmesso alla stessa frequenza del vecchio segnale in bianco e nero, in modo da poter essere ricevuto dai vecchi televisori in bianco e nero, ed i due segnali di crominanza sono trasmessi in bande più strette, a frequenze più elevate; alcuni apparecchi televisivi hanno manopole o controlli, etichettati luminosità colore e saturazione, per controllare questi tre segnali separatamente. Comprendere i concetti di luminanza e crominanza è necessario per rendersi conto di come funzioni la compressione video.

Abbiamo finora considerato il video analogico; passiamo adesso a quello digitale. La più semplice rappresentazione di un video digitale è una sequenza di frame, ciascuno dei quali si compone di una griglia rettangolare di elementi di immagine o pixel. Per il video a colori sono usati 8 bit per pixel per ciascuno dei colori RGB, dando luogo a sedici milioni di colori, che sono abbastanza. L'occhio umano non può infatti distinguere così tanti, e a maggior ragione un numero più elevato.

Per produrre un movimento uniforme, il video digitale, come quello analogico, deve visualizzare almeno 25 frame/s. In ogni caso, poiché i monitor di buona qualità spesso scandiscono lo schermo con immagini memorizzate nella RAM video 75 volte il secondo o più, l'interallacciamento non è necessario. Di conseguenza, tutti i monitor dei computer utilizzano la scansione progressiva. Ricolorare (vale a dire ridisegnare) lo stesso frame tre volte per riga è sufficiente al fine di eliminare lo sfarfallio.

In altre parole, l'uniformità del movimento è determinata dal numero di diverse immagini per secondo, mentre lo sfarfallio è determinato dal numero di volte il secondo in cui lo schermo viene colorato. Questi due parametri sono differenti: un'immagine digitale

gnata a 20 frame/s non mostrerà un movimento a scatti, ma sfarfallerà, perché un frame decadrà dalla retina prima che il successivo appaia, mentre un film con venti diversi frame/s, ciascuno dei quali è disegnato quattro volte in una riga a 80 Hz, non sfarfallerà, ma il movimento apparirà a scatti.

La significatività di questi due parametri diviene chiara qualora si consideri la larghezza di banda richiesta per trasmettere video digitali in rete. Gli attuali monitor utilizzano tutti la proporzione 4:3 fra le dimensioni dello schermo in modo da poter usare tubi catodici economici, prodotti in quantità per il mercato dei televisori; sono configurazioni comuni 640 x 480 (VGA), 800 x 600 (SVGA) e 1024 x 768 (XGA). Un dispositivo XGA con 24 bit per pixel e 25 frame/s, deve ricevere i dati a 472 Mbps: raddoppiare questa frequenza per evitare lo sfarfallio non è divertente. È meglio trasmettere 25 frame/s facendo memorizzare e disegnare al computer ciascun frame per due volte. La trasmissione televisiva non usa questa strategia, poiché i televisori non sono provvisti di memoria, e in ogni caso non è possibile memorizzare in RAM i segnali analogici senza prima convertirli in formato digitale, il che richiederebbe dell'hardware in più. Perciò, l'interallaccinamento è necessario per la trasmissione televisiva, ma non per il video digitale.

## 7.3 Compressione video

Come dovrebbe essere ovvio già da ora, non si può manipolare il materiale multimediale in forma non compressa, in quanto richiederebbe enormi quantità di memoria: la sola speranza è poterlo comprimere in modo rilevante. Per fortuna, negli ultimi decenni la ricerca ha prodotto diverse tecniche di compressione e algoritmi che rendono possibile trasmettere i dati multimediali. In questa sezione studieremo alcuni metodi di compressione dei dati multimediali, in particolare delle immagini. (Per una trattazione più dettagliata: Fluckiger, 1995; Steinmetz e Nahrstedt 1995).

Tutti i sistemi di compressione richiedono due algoritmi: uno per comprimere i dati alla sorgente, il secondo per decomprimere alla destinazione. In letteratura tali algoritmi sono definiti di **codifica** e **decodifica**, e anche noi useremo questa terminologia.

Ci sono, in questi algoritmi, alcune asimmetrie indicative; in primo luogo, per molte applicazioni, un documento multimediale, ad esempio un film, sarà codificato una sola volta (quando è memorizzato sul server multimediale), ma sarà decodificato migliaia di volte (quando lo vedono i diversi clienti). Ciò rende accettabile che l'algoritmo di codifica sia lento e necessiti di hardware costoso, ma l'algoritmo di decodifica dev'essere veloce.

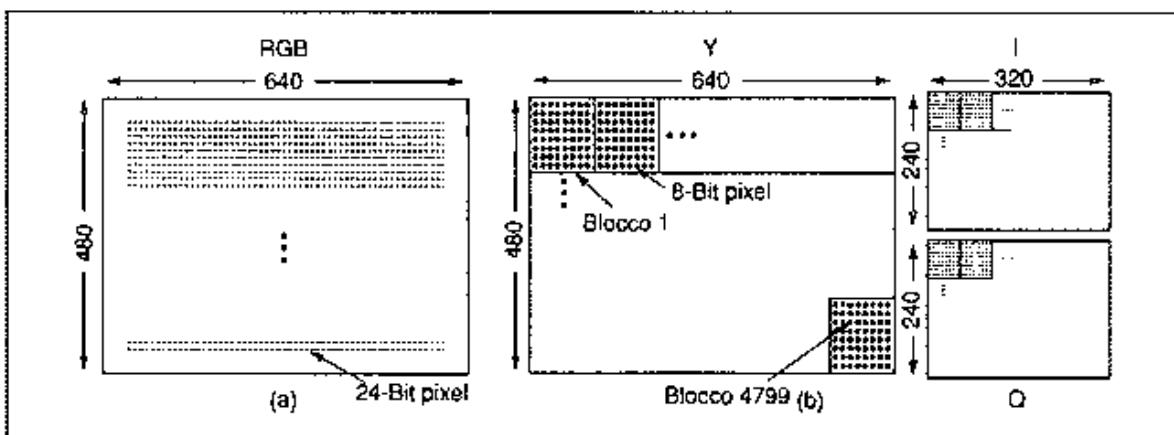


Figura 7.6 (a) Dati di input RGB. (b) Preparazione dei blocchi.

ce, e l'hardware poco costoso. Al contrario, per i multimedia in tempo reale, come una videoconferenza, una codifica lenta è inaccettabile: essa deve avvenire istantaneamente, in tempo reale.

Una seconda asimmetria è che il processo di codifica-decodifica non è necessariamente invertibile: quando un file è compresso, trasmesso e quindi decompresso, l'utente si aspetta di riottenere l'originale, accurato fino all'ultimo bit. Con i multimedia invece, tale esigenza non esiste: di solito, è accettabile ricevere il segnale video, dopo la codifica, in una forma decodificata leggermente diversa dall'originale. Quando l'output decodificato non è esattamente uguale all'input originale, si afferma che il sistema è *lossy* (cioè ha perso parte dell'informazione). Tutti i sistemi di compressione per i multimedia sono lossy, perché forniscono una compressione molto migliore.

### 7.3.1 Lo standard JPEG

Lo standard **JPEG** (Joint Photographic Experts Group, Gruppo di Esperti Fotografici Riuniti) per la compressione a tori continui di immagini fisse (ad esempio fotografie) è stato sviluppato da esperti in fotografia che lavoravano per conto degli organismi di standardizzazione ITU, ISO e IEC. È importante, per i multimedia, perché, in prima approssimazione, lo standard multimediale per le immagini in movimento, **MPEG**, è proprio la codifica JPEG separata di ciascun frame, con alcune caratteristiche in più per la compressione tra frame e la compensazione del movimento. JPEG è definito nello Standard Internazionale 10918. Ha quattro modi e molte opzioni, ma prenderemo in considerazione soltanto il modo per il video RGB a 24 bit, e tralasseremo molti dettagli.

Il passo 1 di codifica di un'immagine con JPEG consiste nella preparazione dei blocchi. Per essere più precisi, si supponga che l'input JPEG sia un'immagine RGB 640 x 480, con 24 bit/pixel, come si vede nella Figura 7.6 (a). Poiché utilizzando la luminanza e la crominanza si ottiene una migliore compressione, la luminanza e due segnali di crominanza sono calcolati dai valori RGB. Per NTSC sono chiamati rispettivamente  $Y, I, Q$ ; per PAL sono chiamati rispettivamente  $Y, U$  e  $V$ , e le formule sono differenti. Di seguito useremo i nomi NTSC, ma l'algoritmo di compressione è lo stesso.

Si costruiscono matrici separate per  $Y, I, Q$ , ciascuna con elementi nell'intervallo tra 0

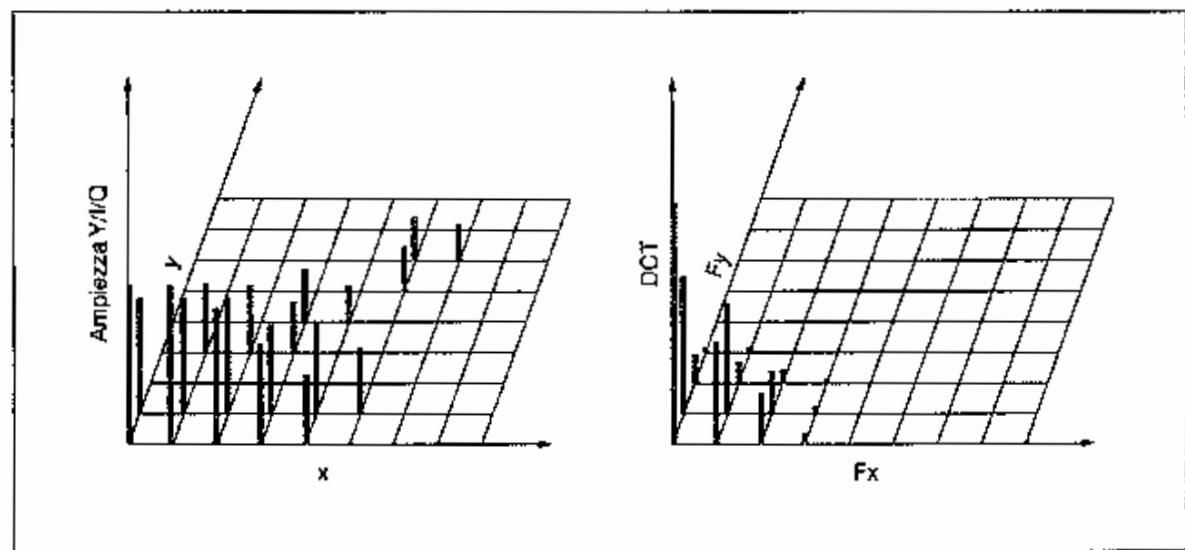


Figura 7.7 (a) Un blocco della matrice Y. (b) I coefficienti DCT.

e 255; successivamente, si effettua la media su quadrati di 4 pixel nelle matrici di  $I$  e  $Q$ , per ridurle a  $320 \times 240$ . Questa riduzione è lossy, ma l'occhio se ne accorge a malapena, in quanto risponde più alla luminanza che alla crominanza, e comprime i dati di un fattore due. Ora, si sottrae 128 da ciascun elemento di tutte e tre le matrici, per porre lo 0 al centro dell'intervallo, infine ciascuna matrice è divisa in blocchi  $8 \times 8$ . La matrice  $Y$  ha 4800 blocchi, le altre due ne hanno 1200 ciascuna, come si vede nella Figura 7.6 (b).

Il passo 2 di JPEG consiste nell'applicare una DCT (Discrete Cosine Transformation, Trasformata Coseno Discreta) a ciascuno dei 7200 blocchi separatamente. L'output di ciascuna DCT è una matrice  $8 \times 8$  di coefficienti DCT, l'elemento DCT (0,0) è il valore medio del blocco, e gli altri elementi esprimono quanta potenza spettrale è presente in ciascuna frequenza spaziale. In teoria una DCT non genera perdita di informazione, ma in pratica utilizzando numeri in virgola mobile e funzioni trascendenti, introduce sempre un certo errore di arrotondamento che produce una piccola perdita di informazione. Normalmente questi elementi diminuiscono rapidamente, con la distanza dall'origine (0,0), come suggerito dalla Figura 7.7 (b).

Una volta che la DCT è completa, JPEG va al passo 3, chiamato quantizzazione, nel quale i coefficienti DCT meno importanti sono eliminati. Questa trasformazione (lossy) viene effettuata dividendo ciascuno dei coefficienti della matrice DCT  $8 \times 8$  per un peso preso da una tabella. Se tutti i pesi sono 1, la trasformazione non fa nulla, ma se i pesi aumentano rapidamente dall'origine, le frequenze spaziali più elevate diminuiscono altrettanto rapidamente.

Un esempio di questo passo è illustrato nella Figura 7.8, in cui si vedono la matrice iniziale DCT, la tabella di quantizzazione ed il risultato ottenuto dividendo ciascun elemento DCT per il corrispondente elemento della tabella di quantizzazione. I valori di quest'ultima non fanno parte dello standard JPEG: ciascun'applicazione deve fornire la propria tabella di quantizzazione, in modo da poter determinare un proprio trade-off fra perdita di informazioni e compressione.

Il passo 4 riduce il valore (0,0) di ciascun blocco (quello nell'angolo a sinistra in alto), riappiattandolo con la quantità di cui differisce dall'elemento corrispondente del blocco precedente. Poiché questi elementi sono le medie dei loro rispettivi blocchi, dovrebbero cambiare lievemente, cosicché, considerando i valori delle differenze, ridurremo la maggior parte di essi a valori piccoli. Nessuna differenza viene calcolata per gli altri valori; i valori (0,0) sono detti componenti DC; gli altri valori AC.

Il passo 5 linearizza i 64 elementi e applica una codifica in lunghezza alla lista. Per-

| Coeffienti DCT |    |    |    |   |   |   |   | Coeffienti quantizzati |    |    |   |   |   |   |   | Tabella di quantizzazione |    |    |    |    |    |    |    |
|----------------|----|----|----|---|---|---|---|------------------------|----|----|---|---|---|---|---|---------------------------|----|----|----|----|----|----|----|
| 150            | 80 | 40 | 14 | 4 | 2 | 1 | 0 | 150                    | 80 | 20 | 4 | 1 | 0 | 0 | 0 | 1                         | 1  | 2  | 4  | 8  | 16 | 32 | 64 |
| 92             | 75 | 36 | 10 | 6 | 1 | 0 | 0 | 92                     | 75 | 18 | 3 | 1 | 0 | 0 | 0 | 1                         | 1  | 2  | 4  | 8  | 16 | 32 | 64 |
| 52             | 38 | 26 | 8  | 7 | 4 | 0 | 0 | 26                     | 19 | 13 | 2 | 1 | 0 | 0 | 0 | 2                         | 2  | 2  | 4  | 8  | 16 | 32 | 64 |
| 12             | 8  | 6  | 4  | 2 | 1 | 0 | 0 | 3                      | 2  | 2  | 1 | 0 | 0 | 0 | 0 | 4                         | 4  | 4  | 4  | 8  | 16 | 32 | 64 |
| 4              | 3  | 2  | 0  | 0 | 0 | 0 | 0 | 1                      | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 8                         | 8  | 8  | 8  | 8  | 16 | 32 | 64 |
| 2              | 2  | 1  | 1  | 0 | 0 | 0 | 0 | 0                      | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 16                        | 16 | 16 | 16 | 16 | 16 | 32 | 64 |
| 1              | 1  | 0  | 0  | 0 | 0 | 0 | 0 | 0                      | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 32                        | 32 | 32 | 32 | 32 | 32 | 32 | 64 |
| 0              | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0                      | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 64                        | 64 | 64 | 64 | 64 | 64 | 64 | 64 |

Figura 7.8 Calcolo dei coefficienti DCT quantizzati.

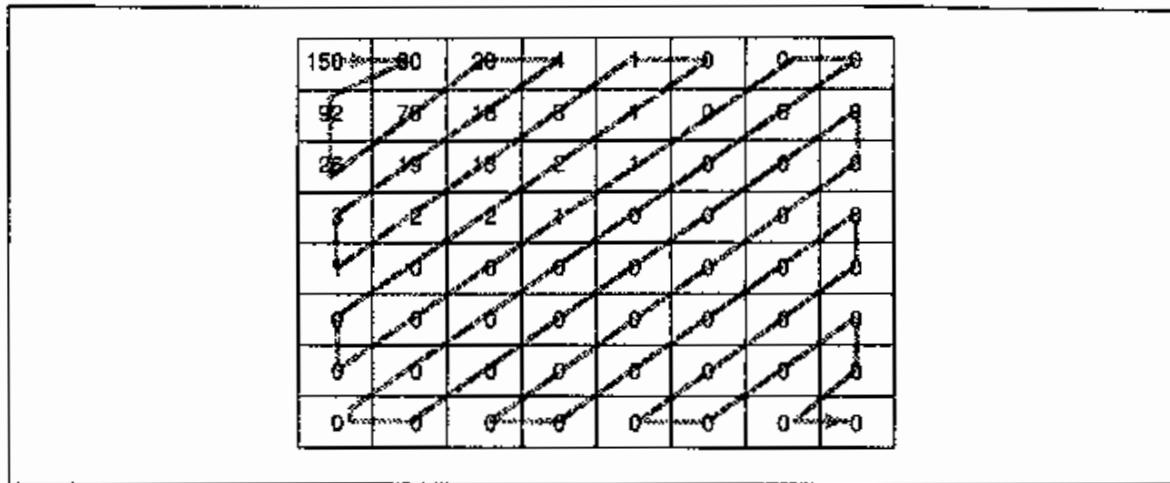


Figura 7.9 L'ordine in cui i valori quantizzati vengono trasmessi.

correndo il blocco da sinistra a destra e quindi dall'alto al basso, non concentreremo gli zeri insieme, e quindi si usa uno schema a zig-zag, come si vede nella Figura 7.9. In questo esempio lo schema a zig-zag produce in ultima analisi 38 zeri consecutivi alla fine della matrice, tale sequenza può essere ridotta ad un singolo valore, che afferma che ci sono 38 zeri.

Ora abbiamo una lista di numeri che rappresenta l'immagine (nello spazio trasformato); il passo 6 utilizza l'algoritmo di Huffman per codificare i numeri per la memorizzazione e la trasmissione.

JPEG può sembrare complicato, proprio perché è complicato. Comunque, poiché spesso produce una compressione di 20:1 o di più, è ampiamente usato. La decodifica di un'immagine JPEG richiede l'esecuzione dell'algoritmo all'indietro; JPEG è, a grandi linee, simmetrico, e richiede circa lo stesso lavoro per codificare e decodificare un'immagine.

### 7.3.2 Lo standard MPEG

Finalmente arriviamo al cuore della materia: gli standard **MPEG** (Motion Picture Experts Group, Gruppo di Esperti di Immagini in Movimento). Questi sono i principali algoritmi utilizzati per comprimere i video, e sono standard internazionali dal 1993. MPEG-1 (Standard Internazionale 11172), è stato progettato per una qualità di uscita da video registratore (352 x 240 per NTSC), a 1,2 Mbps. MPEG-2 (Standard Internazionale 13118) è stato progettato per comprimere i segnali di qualità video televisiva da 4 a 6 Mbps, in modo da poter essere utilizzato nei canali televisivi NTSC o PAL.

Entrambe le versioni si avvantaggiano dei due tipi di ridondanze che esistono nei film: spaziale e temporale. La ridondanza spaziale può essere utilizzata codificando ciascun frame separatamente con JPEG; una compressione ulteriore si ottiene avvalendosi del fatto che frame consecutivi sono spesso quasi identici (ridondanza temporale). Il sistema DV (Digital Video), usato dalle videocamere digitali, utilizza solo uno schema tipo JPEG, perché la codifica dev'essere fatta in tempo reale, ed è molto più veloce codificare ciascun frame separatamente. Le conseguenze di questa decisione si possono vedere nella Figura 7.2: benché le videocamere digitali abbiano una minore velocità di trasmissione dati rispetto ai video non compressi, non raggiungono le prestazioni dello standard MPEG-2. (Per un confronto obiettivo, si noti che le videocamere DV campionano la luminanza con 8 bit, e ciascun segnale di crominanza con 2 bit, ma c'è ancora un fattore di compressione cinque, utilizzando la codifica tipo JPEG).

Per scene dove la telecamera e lo sfondo sono fissi, e uno o due attori si muovono lentamente, quasi tutti i pixel saranno identici da frame a frame; in questo caso si può ottenere un buon risultato semplicemente sottraendo ciascun frame dal precedente, ed eseguendo JPEG sulla differenza. Però, per scene dove la telecamera esegue una panoramica od uno zoom, questa tecnica fallisce; ciò che ci vuole è una modalità di compensazione del movimento, e questo è precisamente ciò che MPEG fa, ed è questa la sua principale differenza rispetto a JPEG.

L'uscita di MPEG-2 si compone di tre diversi tipi di frame, che devono essere elaborati dal programma di visualizzazione:

1. I (frame intracodificati): immagini fisse auto contenute codificate tramite JPEG.
2. P (frame predittivi): differenze blocco per blocco con l'ultimo frame.
3. B (frame bidirezionali): differenze rispetto al frame precedente e a quello successivo.

I frame I sono immagini fisse codificate tramite JPEG, che utilizzano la luminanza a piena risoluzione, e la crominanza a metà risoluzione, lungo ciascun asse. I frame I devono apparire periodicamente in uscita, per tre ragioni: in primo luogo, i MPEG possono essere usati per la trasmissione televisiva, con gli spettatori che si sintonizzano quando lo desiderano; se tutti i frame dipendessero dai precedenti fino al primo, chiunque abbia perso il primo frame non potrebbe mai decodificare i successivi, e ciò non permetterebbe agli spettatori di sintonizzarsi dopo che il film è iniziato. In secondo luogo, se un frame fosse ricevuto erroneamente non sarebbe possibile decodificare i successivi; senza i frame I, inoltre, mentre si effettua un avanti o indietro veloce, il decodificatore dovrebbe calcolare ciascun frame su cui è passato, se vuole conoscere il valore di quello su cui si è fermato, ma con i frame I è possibile saltare avanti o indietro fino a trovare un frame I, ed iniziare da quel punto la visualizzazione. Per queste ragioni i frame I sono inseriti in uscita una o due volte il secondo.

I frame P, al contrario, codificano le differenze tra frame. Sono basati sull'idea dei macroblocchi, che coprono uno spazio di  $16 \times 16$  pixel in luminanza, e di  $8 \times 8$  pixel in crominanza. Un macroblocco è codificato cercando nel precedente frame il macroblocco stesso, o qualcosa che differisca solo di poco da esso.

Un esempio di situazione in cui i frame P sarebbero utili è presentato nella Figura 7.10. Si vedono tre frame consecutivi con lo stesso sfondo, ma con una persona in posizioni diverse. I macroblocchi contenenti lo sfondo della scena corrisponderanno, mentre quelli che contengono la persona saranno spostati di una quantità non nota, e occorrerà rintracciarli.

Lo standard MPEG non specifica come e quanto in profondità cercare, o quanto precisa debba essere una corrispondenza perché sia considerata tale; questo è compito delle

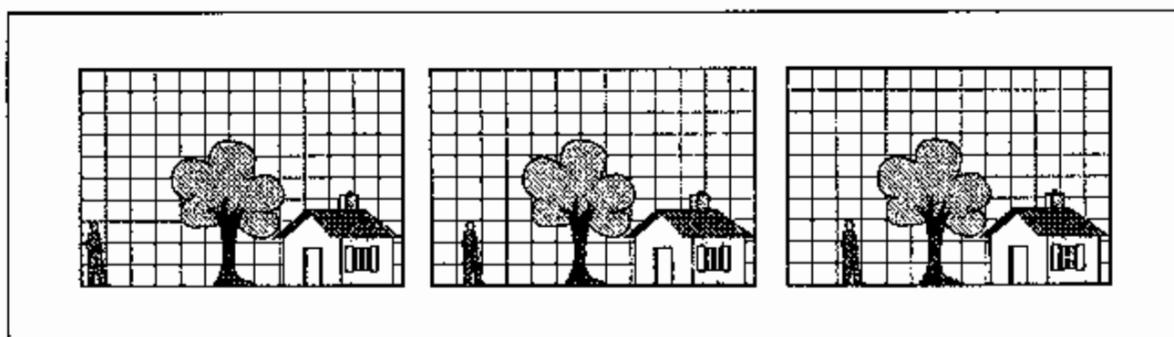


Figura 7.10 Tre frame video consecutivi.

diverse implementazioni. Per esempio, un'implementazione potrebbe cercare un macroblocco nella posizione corrente nel frame precedente, e in tutte le altre posizioni con uno scostamento  $\pm \Delta x$  nella direzione  $x$ , e  $\pm \Delta y$  nella direzione  $y$ . Per ciascuna posizione si potrebbe calcolare il numero delle corrispondenze nella matrice di luminanza, e la posizione col punteggio più alto sarebbe dichiarata vincente, a condizione che sia sopra ad una soglia prefissata; diversamente il macroblocco sarebbe considerato perduto. Naturalmente sono anche possibili algoritmi assai più sofisticati.

Se si trova un macroblocco, lo si codifica prendendo la differenza col suo valore nel frame precedente (per la luminanza ed entrambe le crominanze); tali matrici differenza vengono poi sottoposte alla codifica JPEG, ed il valore del macroblocco in uscita è quindi il vettore di movimento (quanto il macroblocco si sia spostato dalla sua precedente posizione in ciascuna direzione), seguito dalle differenze rispetto a quello del precedente frame, codificate secondo lo standard JPEG. Se il macroblocco non è localizzato nel frame precedente, il valore corrente è codificato tramite JPEG, come se fosse un frame I.

I frame B sono simili ai P, con la differenza che il macroblocco di riferimento può essere sia nel frame precedente sia in quello successivo, o in un frame I, oppure in un frame P; questa ulteriore libertà permette una migliore compensazione del movimento, ed è anche utile quando gli oggetti passano davanti o dietro ad altri oggetti. Ad esempio, in una partita di baseball, quando l'uomo in terza base lancia la palla a quello in prima base, possiamo avere alcuni frame dove la palla nasconde la testa dell'uomo che si muove sullo sfondo, in seconda base. Nel frame successivo la testa può essere parzialmente visibile a sinistra della palla, con la successiva approssimazione della testa che deriva dal frame seguente, quando la palla è ormai passata oltre la testa. I frame B permettono ad un frame di basarsi su di un frame successivo.

Per effettuare la codifica di un frame B, il codificatore ha bisogno di mantenere contemporaneamente in memoria tre frame decodificati: quello passato, quello corrente, e quello futuro. Per semplificare la decodifica, i frame devono essere presenti nel flusso MPEG in ordine di dipendenza, piuttosto che in ordine di visualizzazione. Quindi, anche con una temporizzazione perfetta, quando un video è visualizzato su una rete, si richiede la bufferizzazione sulla macchina dell'utente, in modo da riordinare i frame per visualizzarli correttamente. A causa di questa differenza fra l'ordine di dipendenza e quello di visualizzazione, cercare di eseguire un film all'indietro non sarà possibile, senza ricorrere a considerevoli meccanismi di bufferizzazione e ad algoritmi complessi.

## 7.4 Schedulazione dei processi multimediali

I sistemi operativi che supportano i multimedia differiscono da quelli tradizionali per tre aspetti principali: la schedulazione dei processi, il file system e la schedulazione del disco. Inizieremo a trattare la schedulazione dei processi, quindi tratteremo gli altri aspetti nelle sezioni seguenti.

### 7.4.1 Schedulazione di processi omogenei

Il tipo più semplice di video server è quello che supporta la visualizzazione di un numero fisso di film, tutti con la medesima frequenza dei frame, risoluzione video, frequenza di trasmissione dati ed altri parametri. In questa situazione, un algoritmo di schedulazione semplice, ma efficace, è il seguente: per ciascun film esiste un singolo processo (o thread) il cui compito consiste nel leggere il film dal disco un frame alla volta, e

quindi trasmetterlo all'utente. Poiché tutti i processi sono ugualmente importanti, hanno la stessa quantità di lavoro per frame da svolgere, e si bloccano quando hanno terminato l'elaborazione del frame corrente, la schedulazione round-robin è adatta allo scopo. È necessario aggiungere all'algoritmo di schedulazione standard soltanto un meccanismo di temporizzazione, per essere certi che ogni processo sia eseguito alla frequenza corretta.

È possibile ottenere una temporizzazione appropriata con un clock principale che scatti, ad esempio, 30 volte il secondo (per NTSC). Ad ogni scatto, tutti i processi sono eseguiti in sequenza nello stesso ordine, e quando un processo ha completato il proprio compito, esegue una chiamata di sistema `suspend` che rilascia la CPU finché il clock principale scatta di nuovo. Quando ciò accade, tutti i processi sono nuovamente eseguiti nello stesso ordine; se il numero dei processi è abbastanza piccolo da poter eseguire tutto il lavoro in un tempo di frame, la schedulazione round-robin è sufficiente.

## 7.4.2 Schedulazione generale in tempo reale

Disgraziatamente, questo modello si può applicare, in pratica, molto di rado. Il numero degli utenti cambia, poiché gli spettatori vanno e vengono, le dimensioni dei frame variano di molto, a causa delle caratteristiche della compressione video (i frame I sono molto più grandi dei frame P, o B), e film diversi possono avere risoluzioni diverse; quindi è possibile dover eseguire diversi processi a frequenze diverse, con quantità di lavoro diverse, e diverse scadenze entro le quali completare il lavoro stesso.

Queste considerazioni conducono ad un modello diverso: più processi che competono per la CPU, ciascuno col proprio carico di lavoro e le proprie scadenze temporali. Nei modelli descritti in seguito, supporremo che il sistema conosca la frequenza cui eseguire ciascun processo, quanto lavoro debba compiere, e la successiva scadenza temporale; anche la schedulazione del disco è un problema, di cui però ci occuperemo più avanti. La schedulazione di più processi in competizione, alcuni dei quali, o tutti, abbiano scadenze temporali da rispettare, è definita **schedulazione in tempo reale**.

Come esempio di ambiente in cui lavori uno schedulatore multimediale in tempo reale, si considerino i tre processi A, B, C della Figura 7.11: il processo A viene eseguito ogni

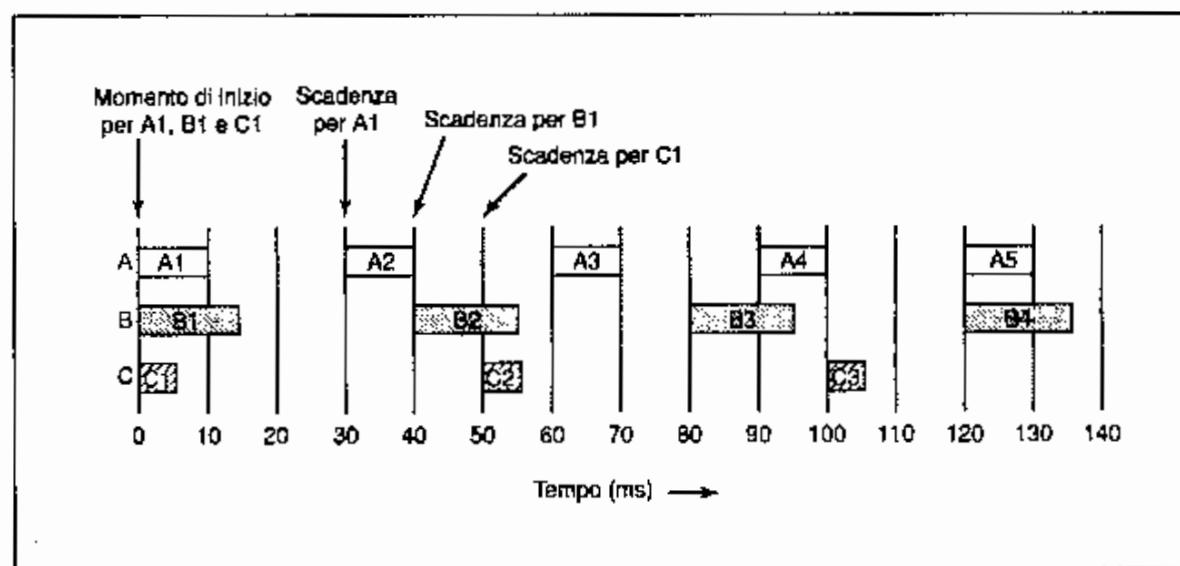


Figura 7.11 Tre processi periodici che visualizzano ciascuno un film. Le frequenze dei frame e i tempi di elaborazione per frame sono diversi per ciascun film.

30 ms (all'incirca la velocità NTSC), e ciascun frame richiede 10 ms di tempo di CPU. In assenza di competizione, sarebbe eseguito nei periodi A1, A2, A3, ecc., ciascuno 30 ms dopo il precedente. Ogni periodo di CPU gestisce un frame e ha una scadenza, cioè deve finire prima chi inizi il successivo.

Nella Figura 7.11 sono illustrati anche i due processi B e C. B viene eseguito 25 volte/s (ad esempio, PAL), C 20 volte/s (ad esempio, un flusso NTSC o PAL rallentato per utenti connessi al video server con larghezza di banda ridotta). Il tempo di calcolo per frame è di 15 ms e 5 ms, per B e C rispettivamente, per rendere il problema di schedulazione più generale.

A questo punto, il problema è come schedulare A, B e C per essere certi che rispettino ciascuno le proprie scadenze temporali. Prima ancora di cercare un algoritmo di schedulazione, dobbiamo però valutare se questo insieme di processi sia effettivamente schedulabile. Si ricordi (Paragrafo 2.5.4) che se il processo  $i$  ha un periodo di  $P_i$  ms e richiede  $C_i$  ms di tempo di CPU per frame, il sistema è schedulabile se e solo se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

dove  $m$  è il numero dei processi, in questo caso tre. Si noti che  $P_i/C_i$  è proprio la frazione del tempo di CPU utilizzato dal processo  $i$ : nell'esempio della Fig. 7.11, A consuma i 10/30 di tempo della CPU, B i 15/40 e C i 5/50, e la somma di queste frazioni ammonta a 0,808 del tempo di CPU, e quindi il sistema dei processi è schedulabile.

Finora abbiamo supposto che esista un processo per flusso; in realtà, potrebbero essercene due, o più, per processo, ad esempio uno per l'audio e il secondo per il video, ed è possibile eseguirli a frequenze diverse, consumando quantità diverse di tempo di CPU per periodo. Aggiungere processi audio non comporta tuttavia un cambiamento del modello generale, poiché assumiamo che esistano  $m$  processi, ciascuno eseguito ad una frequenza determinata, con una quantità fissa di lavoro per ciascun periodo di CPU utilizzato.

In alcuni sistemi in tempo reale, i processi sono prerilasciabili, e in altri no. Nei sistemi multimediali, i processi sono generalmente prerilasciabili: un processo la cui scadenza temporale sia a rischio, può interrompere il processo in esecuzione prima che esso concluda l'elaborazione del proprio frame; quando ha terminato, il processo precedente può continuare. Questo comportamento, come abbiamo visto in precedenza, è tipico della multiprogrammazione. Studieremo algoritmi di schedulazione in tempo reale prerilasciabili, perché non ci sono controindicazioni nell'utilizzarli nei sistemi multimediali, e garantiscono prestazioni migliori di quelli non prerilasciabili. Il solo elemento da considerare è che, se un buffer di trasmissione viene riempito un po' alla volta, dovrà essere pieno entro la scadenza, in modo da poterlo inviare all'utente con una sola operazione; in caso contrario, avremmo immagini disturbate da un tremolio.

Gli algoritmi in tempo reale possono essere statici o dinamici. Quelli statici assegnano a ciascun processo una priorità determinata in precedenza, ed effettuano una schedulazione con prerilascio e con priorità, utilizzando le priorità stesse. Gli algoritmi dinamici non hanno priorità prefissate. Studieremo un esempio per ciascun tipo.

### 7.4.3 Schedulazione a frequenza monotona

Il classico algoritmo statico di schedulazione in tempo reale per processi periodici e prerilasciabili è RMS (Rate Monotonic Scheduling, schedulazione a frequenza monotona), (Liu e Layland, 1973), utilizzabile per processi che soddisfino le seguenti condizioni:

1. Ciascun processo periodico va completato entro il suo periodo di tempo.

2. Nessun processo è dipendente dagli altri.
3. Ciascun processo abbisogna della stessa quantità di tempo di CPU per ogni periodo di esecuzione.
4. I processi non periodici non hanno scadenze temporali.
5. Il prerilascio dei processi avviene istantaneamente e senza sovraccarico di lavoro per il sistema.

Le prime quattro condizioni sono ragionevoli; l'ultima, ovviamente, non lo è, ma rende più semplice la modellazione del sistema. RMS funziona assegnando a ciascun processo una priorità prefissata uguale alla frequenza cui deve essere eseguito: ad esempio, un processo che debba essere eseguito ogni 30 ms (33 volte/s) acquisisce priorità 33; un processo da eseguire ogni 40 ms (25 volte/s) acquisisce priorità 25, mentre un processo da eseguire ogni 50 ms (20 volte/s) acquisisce priorità 20. Le priorità sono quindi lineari rispetto alla frequenza (numero di volte al secondo in cui il processo è eseguito), per questa ragione, esso è definito a frequenza monotona. Durante l'esecuzione, lo scheduler esegue sempre il processo pronto a più alta priorità, prerilasciando, se necessario, il processo in esecuzione. Liu e Layland hanno dimostrato che RMS è ottimale rispetto alla classe di algoritmi di schedulazione statici.

La Figura 7.12 illustra il funzionamento dell'algoritmo di schedulazione a frequenza monotona nell'esempio della Figura 7.11. I processi A, B e C hanno rispettivamente priorità statiche 33, 25 e 20, cioè, quando A deve andare in esecuzione, prerilascia ogni altro processo; B può prerilasciare C, ma non A. Per andare in esecuzione, C deve aspettare finché la CPU diviene inattiva.

Nella Figura 7.12, inizialmente tutti i processi sono pronti per l'esecuzione. Viene scelto A, quello a priorità più elevata, e resta in esecuzione finché termina, dopo 10 ms, come mostrato nella linea RMS. Terminato A, B e C sono eseguiti nell'ordine; l'esecuzione di questi processi richiede 30 ms, in modo che, al termine di C, A viene nuovamente eseguito. Questa rotazione prosegue, finché il sistema diventa inattivo al tempo  $t = 70$ .

Al tempo  $t = 80$ , B diviene pronto, e passa in esecuzione; a  $t = 90$ , il processo a maggior priorità, A, diviene pronto, prerilascia B e passa in esecuzione fino al termine, al tempo  $t = 100$ . A quel punto il sistema può scegliere di terminare B, oppure iniziare C, e sceglie il processo a maggior priorità, cioè B.

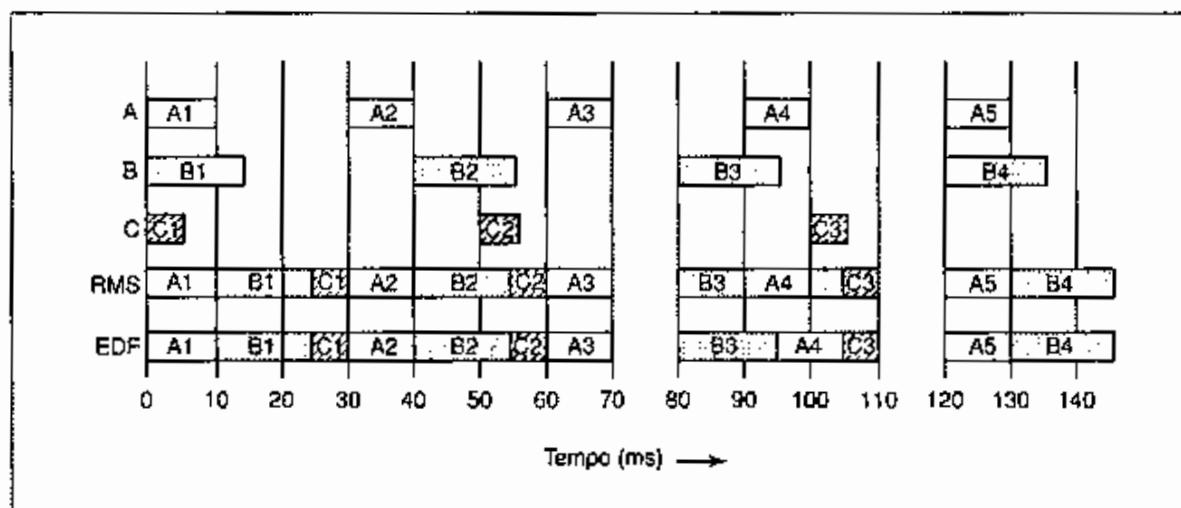


Figura 7.12 Un esempio di schedulazione in tempo reale RMS e EDF.

#### 7.4.4 Schedulazione con priorità alla scadenza più vicina

Un altro diffuso algoritmo di schedulazione in tempo reale è **Earliest Deadline First** (schedulazione con priorità alla scadenza più vicina). EDF è un algoritmo dinamico, e non richiede che i processi siano periodici, come accade per l'algoritmo a frequenza monotona, né che abbiano lo stesso tempo di esecuzione per periodo di CPU, come richiede invece RMS: quando un processo ha bisogno di tempo di CPU, annuncia la sua presenza e la sua scadenza temporale. Lo schedulatore mantiene una lista dei processi eseguibili, ordinata rispetto alla scadenza temporale; l'algoritmo esegue il primo processo della lista, cioè quello con la scadenza temporale più vicina. Quando un nuovo processo diviene pronto, il sistema controlla se la sua scadenza precede quella del processo correntemente in esecuzione; in caso affermativo, il nuovo processo prelascia quello corrente.

La Figura 7.12 presenta un esempio di EDF: inizialmente tutti i processi sono pronti, ed eseguiti nell'ordine delle loro scadenze temporali: A deve finire entro  $t = 30$ , B entro  $t = 40$ , e C entro  $t = 50$ , quindi A ha la scadenza più ravvicinata, e viene eseguito per primo. Fino a  $t = 90$ , le scelte sono le stesse di RMS: per  $t = 90$ , A diviene nuovamente pronto, e la sua scadenza è a  $t = 120$ , la stessa di B. Lo schedulatore potrebbe scegliere di eseguire l'uno o l'altro, ma poiché il prelascio di B ha un costo non nullo, è meglio permettere a B di continuare l'esecuzione.

Per sfatare l'idea che RMS e EDF diano sempre gli stessi risultati, consideriamo ora l'esempio della Figura 7.13. Qui, i periodi di A, B e C sono gli stessi di prima, ma A ha bisogno di 15 ms di tempo di CPU, invece di soli 10 ms. Il test di schedulabilità calcola che l'utilizzo della CPU sia pari a  $0,500 + 0,375 + 0,100 = 0,975$ : solo il 2,5% della CPU resta inutilizzato, ma, in teoria, la CPU non è sovrautilizzata, e dovrebbe essere possibile trovare una schedulazione legale.

Con RMS le priorità dei tre processi sono ancora 33, 25 e 20, poiché contano solo i periodi, e non il tempo di esecuzione; in questo caso, B1 non finisce sino a  $t = 30$ , e in quel momento, A è di nuovo pronto per l'esecuzione. Nel momento in cui A termina, a  $t = 45$ , B è di nuovo pronto, ed avendo maggior priorità rispetto a C, passa in esecuzione e C perde la sua scadenza temporale: RMS fallisce.

Analizziamo ora come EDF gestisce questo caso: a  $t = 30$ , c'è una competizione fra A2 e C1. Poiché la scadenza di C1 è 50, e quella di A2 è 60, viene schedulato C. In RMS, invece, prevale la maggior priorità di A.

A  $t = 90$ , A diviene pronto per la quarta volta. La scadenza di A è la stessa del processo corrente (120), e lo schedulatore può scegliere di prelasciare o meno. Come nel caso

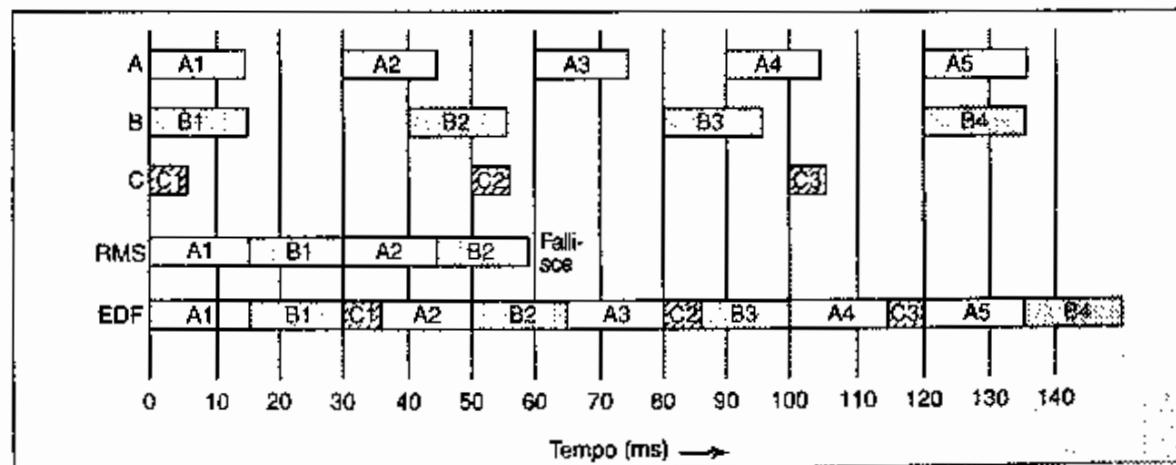


Figura 7.13 Un altro esempio di schedulazione in tempo reale con RMS e EDF.

precedente, è meglio non farlo, se non è necessario, permettendo a B3 di terminare.

Nell'esempio della Figura 7.13, la CPU è occupata al 100% fino a  $t = 150$ ; prima o poi, si avrà un intervallo di non utilizzo, poiché la CPU è utilizzata solo al 97,5%. Siccome tutti i tempi di inizio e fine sono multipli di 5 ms, l'intervallo sarà di 5 ms, e per ottenere il tempo di inutilizzo richiesto del 2,5%, l'intervallo di 5 ms dovrà cadere ogni 200 ms, ed è questa la ragione per cui esso non appare nella Figura 7.13.

È interessante chiedersi perché RMS fallisca: fondamentalmente, utilizzando priorità statiche, esso funziona solo se l'utilizzo della CPU non è troppo elevato. Liu e Layland (1973) hanno dimostrato che, per ogni sistema di processi periodici, se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

allora è garantito il funzionamento di RMS. Per 3, 4, 5, 10, 20 e 100, le massime utilizzazioni permesse sono 0,780, 0,757, 0,743, 0,718, 0,705 e 0,696. Per  $m$  che tende ad infinito, l'utilizzo massimo tende asintoticamente a  $\ln 2$ ; in altre parole, Liu e Layland hanno dimostrato che per tre processi RMS funziona sempre, se l'utilizzazione della CPU è uguale o minore di 0,780 (nel nostro primo esempio, era 0,808, ed RMS funzionava, ma si tratta solo di una caso fortunato: con periodi e tempi di esecuzione diversi, una utilizzazione pari a 0,808 potrebbe far fallire l'algoritmo). Nel secondo esempio, l'utilizzo della CPU era così elevato (0,975), da non lasciare alcuna speranza nel funzionamento di RMS.

Al contrario, EDF funziona sempre, per qualunque insieme di processi schedulabile, e può raggiungere il 100% di utilizzo della CPU, al prezzo, però, di un algoritmo più complesso. Quindi, in un video server reale, se l'utilizzo della CPU è sotto il limite RMS, questo algoritmo si può utilizzare, altrimenti, si dovrebbe scegliere EDF.

## 7.5 Paradigmi per file system multimediali

Ora che abbiamo esaurito la schedulazione dei processi nei sistemi multimediali, continuiamo il nostro studio con i file system multimediali, che usano un paradigma diverso rispetto a quelli tradizionali. Riassumeremo in primo luogo l'I/O tradizionale dei file, quindi sposteremo la nostra attenzione sul modo in cui sono organizzati i file server multimediali. Per accedere ad un file, dapprima il processo effettua una chiamata di sistema open. Se questo accade, al chiamante viene dato un tipo di token, chiamato descrittore di file in UNIX, oppure gestore (handle) in Windows, perché sia usato nelle chiamate future. A quel punto, il processo può effettuare una chiamata di sistema read, fornendo come parametri il token, l'indirizzo del buffer, ed il numero dei byte; il sistema operativo restituisce poi i dati richiesti nel buffer; si possono effettuare sino alla fine del processo chiamate read successive; in conclusione, una chiamata close chiude il file e rilascia le sue risorse.

Questo modello non funziona bene per i multimedia, tenendo conto che è necessario un comportamento in tempo reale, e funziona particolarmente male, se si vuole visualizzare file multimediali provenienti da un video server remoto. Un problema consiste nel fatto che l'utente deve fare le chiamate read spaziate nel tempo con buona precisione; in secondo luogo, il video server dev'essere in grado di fornire i blocchi di dati senza ritardo, cosa difficile quando le richieste arrivano senza essere pianificate, e non sono state riservate in anticipo le adeguate risorse.

Per risolvere questi problemi, i file server multimediali utilizzano un paradigma completamente differente: funzionano come VCR (Video Cassette Recorder, videoregistratori). Per leggere un file multimediale, un processo utente effettua una chiamata di sistema

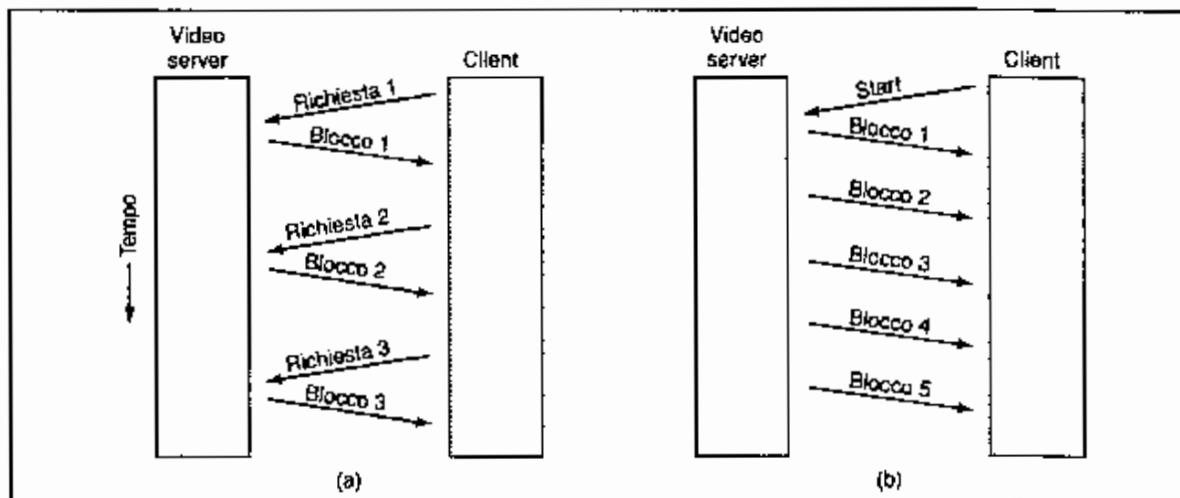


Figura 7.14 (a) Un pull server. (b) Un push server.

start, specificando il file da leggere, e diversi altri parametri, ad esempio quali tracce audio o quali sottotitoli usare. A quel punto, il video server invia frame alla velocità richiesta; è compito dell'utente gestirli alla velocità cui arrivano; se poi l'utente si stancherà del film, la chiamata di sistema stop termina il flusso. I file server con questo modello di flusso dati sono spesso definiti **push server** (perché mandano attivamente dati all'utente), in contrasto con i tradizionali **pull server**, dove l'utente deve prendere i dati in un blocco alla volta, chiamando ripetutamente una read per ottenere un blocco dopo l'altro. La Figura 7.14 illustra la differenza fra questi due modelli.

### 7.5.1 Funzioni di controllo VCR

La maggioranza dei video server implementa anche funzioni di controllo standard VCR, che comprendono la pausa, l'avanti veloce e il riavvolgimento. La pausa è abbastanza semplice; l'utente manda un messaggio al video server dicendogli di fermarsi, ed esso memorizza quale sia il frame successivo. Quando l'utente chiede al server di riprendere, continuerà da dove si era fermato.

Tuttavia, c'è una complicazione. Per ottenere prestazioni accettabili, il server può riservare risorse come larghezza di banda del disco e buffer di memoria per ciascun flusso di uscita. Continuare a mantenere attivi i flussi mentre un film è in pausa, costituisce uno spreco di risorse, in particolare se l'utente sta facendo due passi in cucina per cercare, scongelare, informare e mangiarsi una pizza surgelata (specialmente se è molto grossa). Naturalmente, le risorse si possono rilasciare facilmente nel caso di una pausa, rischiando tuttavia che, quando l'utente cerca di riprendere l'esecuzione, esse non siano disponibili.

Un riavvolgimento completo è veramente facile da gestire: il solo compito del server consiste nell'annotare che il prossimo frame da spedire è lo 0. Come potrebbe essere più semplice? Al contrario, avanti veloce e indietro veloce (visualizzando durante il riavvolgimento) sono procedimenti assai più spinosi. Se non fosse per la compressione, un modo per andare avanti a velocità 10 x, sarebbe visualizzare un frame ogni dieci, ed andare avanti ad una velocità 20 x, richiederebbe visualizzarne uno ogni venti. Infatti, in assenza di compressione, andare avanti o indietro a qualsiasi velocità è semplice: per eseguire ad una velocità  $k$  volte quella normale, è sufficiente visualizzare ogni  $k$ -esimo frame, e per andare indietro ad una velocità  $k$  volte quella normale, si può fare la stessa cosa nella direzione opposta. Questo approccio funziona ugualmente bene per i push server e per i pull server.

precedente, è meglio non farlo, se non è necessario, permettendo a B3 di terminare.

Nell'esempio della Figura 7.13, la CPU è occupata al 100% fino a  $t = 150$ ; prima o poi, si avrà un intervallo di non utilizzo, poiché la CPU è utilizzata solo al 97,5%. Siccome tutti i tempi di inizio e fine sono multipli di 5 ms, l'intervallo sarà di 5 ms, e per ottenere il tempo di inutilizzo richiesto del 2,5%, l'intervallo di 5 ms dovrà cadere ogni 200 ms, ed è questa la ragione per cui esso non appare nella Figura 7.13.

È interessante chiedersi perché RMS fallisca: fondamentalmente, utilizzando priorità statiche, esso funziona solo se l'utilizzo della CPU non è troppo elevato. Liu e Layland (1973) hanno dimostrato che, per ogni sistema di processi periodici, se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

allora è garantito il funzionamento di RMS. Per 3, 4, 5, 10, 20 e 100, le massime utilizzazioni permesse sono 0,780, 0,757, 0,743, 0,718, 0,705 e 0,696. Per  $m$  che tende ad infinito, l'utilizzo massimo tende asintoticamente a  $\ln 2$ ; in altre parole, Liu e Layland hanno dimostrato che per tre processi RMS funziona sempre, se l'utilizzazione della CPU è uguale o minore di 0,780 (nel nostro primo esempio, era 0,808, ed RMS funzionava, ma si tratta solo di una caso fortunato: con periodi e tempi di esecuzione diversi, una utilizzazione pari a 0,808 potrebbe far fallire l'algoritmo). Nel secondo esempio, l'utilizzo della CPU era così elevato (0,975), da non lasciare alcuna speranza nel funzionamento di RMS.

Al contrario, EDF funziona sempre, per qualunque insieme di processi schedulabile, e può raggiungere il 100% di utilizzo della CPU, al prezzo, però, di un algoritmo più complesso. Quindi, in un video server reale, se l'utilizzo della CPU è sotto il limite RMS, questo algoritmo si può utilizzare, altrimenti, si dovrebbe scegliere EDF.

## 7.5 Paradigmi per file system multimediali

Ora che abbiamo esaurito la schedulazione dei processi nei sistemi multimediali, continuiamo il nostro studio con i file system multimediali, che usano un paradigma diverso rispetto a quelli tradizionali. Riassumeremo in primo luogo l'I/O tradizionale dei file, quindi sposteremo la nostra attenzione sul modo in cui sono organizzati i file server multimediali. Per accedere ad un file, dapprima il processo effettua una chiamata di sistema open. Se questo accade, al chiamante viene dato un tipo di token, chiamato descrittore di file in UNIX, oppure gestore (handle) in Windows, perché sia usato nelle chiamate future. A quel punto, il processo può effettuare una chiamata di sistema read, fornendo come parametri il token, l'indirizzo del buffer, ed il numero dei byte; il sistema operativo restituisce poi i dati richiesti nel buffer; si possono effettuare sino alla fine del processo chiamate read successive; in conclusione, una chiamata close chiude il file e rilascia le sue risorse.

Questo modello non funziona bene per i multimedia, tenendo conto che è necessario un comportamento in tempo reale, e funziona particolarmente male, se si vuole visualizzare file multimediali provenienti da un video server remoto. Un problema consiste nel fatto che l'utente deve fare le chiamate read spaziate nel tempo con buona precisione; in secondo luogo, il video server dev'essere in grado di fornire i blocchi di dati senza ritardo, cosa difficile quando le richieste arrivano senza essere pianificate, e non sono state riservate in anticipo le adeguate risorse.

Per risolvere questi problemi, i file server multimediali utilizzano un paradigma completamente differente: funzionano come VCR (Video Cassette Recorder, videoregistratori). Per leggere un file multimediale, un processo utente effettua una chiamata di sistema

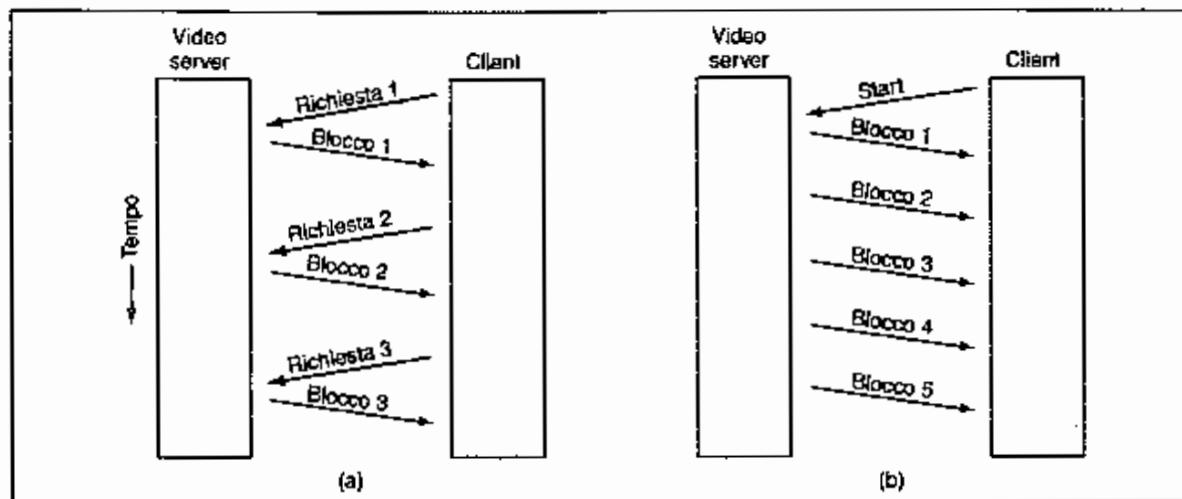


Figura 7.14 (a) Un pull server. (b) Un push server.

start, specificando il file da leggere, e diversi altri parametri, ad esempio quali tracce audio o quali sottotitoli usare. A quel punto, il video server invia frame alla velocità richiesta; è compito dell'utente gestirli alla velocità cui arrivano; se poi l'utente si stancherà del film, la chiamata di sistema stop termina il flusso. I file server con questo modello di flusso dati sono spesso definiti **push server** (perché mandano attivamente dati all'utente), in contrasto con i tradizionali **pull server**, dove l'utente deve prendere i dati in un blocco alla volta, chiamando ripetutamente una **read** per ottenere un blocco dopo l'altro. La Figura 7.14 illustra la differenza fra questi due modelli.

### 7.5.1 Funzioni di controllo VCR

La maggioranza dei video server implementa anche funzioni di controllo standard VCR, che comprendono la pausa, l'avanti veloce e il riavvolgimento. La pausa è abbastanza semplice; l'utente manda un messaggio al video server dicendogli di fermarsi, ed esso inmemorizza quale sia il frame successivo. Quando l'utente chiede al server di riprendere, continuerà da dove si era fermato.

Tuttavia, c'è una complicazione. Per ottenere prestazioni accettabili, il server può riservare risorse come larghezza di banda del disco e buffer di memoria per ciascun flusso di uscita. Continuare a mantenere attivi i flussi mentre un film è in pausa, costituisce uno spreco di risorse, in particolare se l'utente sta facendo due passi in cucina per cercare, scongelare, informare e mangiarsi una pizza surgelata (specialmente se è molto grossa). Naturalmente, le risorse si possono rilasciare facilmente nel caso di una pausa, rischiando tuttavia che, quando l'utente cerca di riprendere l'esecuzione, esse non siano disponibili.

Un riavvolgimento completo è veramente facile da gestire: il solo compito del server consiste nell'annotare che il prossimo frame da spedire è lo 0. Come potrebbe essere più semplice? Al contrario, avanti veloce e indietro veloce (visualizzando durante il riavvolgimento) sono procedimenti assai più spinosi. Se non fosse per la compressione, un modo per andare avanti a velocità 10 x, sarebbe visualizzare un frame ogni dieci, ed andare avanti ad una velocità 20 x, richiederebbe visualizzarne uno ogni venti. Infatti, in assenza di compressione, andare avanti o indietro a qualsiasi velocità è semplice: per eseguire ad una velocità  $k$  volte quella normale, è sufficiente visualizzare ogni  $k$ -esimo frame, e per andare indietro ad una velocità  $k$  volte quella normale, si può fare la stessa cosa nella direzione opposta. Questo approccio funziona ugualmente bene per i push server e per i pull server.

La compressione rende il movimento rapido più complicato. Con un video registrato DV, dove ciascun frame è compresso indipendentemente da tutti gli altri, è possibile utilizzare questa strategia, a patto di poter trovare velocemente il frame necessario. Poiché il rapporto di compressione di ciascun frame è diverso, in quanto dipende dal suo contenuto, ogni frame ha anche una diversa dimensione, quindi non è possibile saltare in avanti di  $k$  frame nel file, facendo un calcolo numerico. Inoltre la compressione audio è indipendente da quella video, quindi per ciascun video frame visualizzato in modalità *avanti veloce* dev'essere anche localizzato il frame audio corretto (sempre che il suono non sia spento quando si esegue a velocità maggiore del normale). Quindi l'avanti veloce per un file DV richiede un indice che permetta di localizzare rapidamente i frame; questo però, almeno in teoria, è fattibile.

Con MPEG questo schema non funziona, nemmeno in teoria, a causa dell'uso dei frame I, P e B. Saltando avanti di  $k$  frame (sempre che ciò sia possibile), si potrebbe finire su un frame P, basato su un frame I, che era stato saltato. Senza il frame di base, un frame P è inutile, perché contiene proprio le variazioni incrementali dal frame che ci manca; MPEG richiede l'esecuzione sequenziale dei file.

Un altro modo di affrontare il problema consiste nel provare davvero ad eseguire il file sequenzialmente a velocità 10 x, il che richiede di trasmettere i dati dal disco ad una velocità 10 x. A quel punto, il server potrebbe provare a decomprimere i frame (come di solito non fa), trovare il frame necessario (uno ogni 10), ricomprimendolo come frame I. Questo procedimento genera però un pesante carico sul server, il quale deve anche comprendere il formato di compressione, cosa di cui di norma non si preoccupa.

L'alternativa di spedire tutti i dati in rete all'utente, e lasciare che i frame corretti siano selezionati, richiede che la rete funzioni ad una velocità 10 x, il che è possibile, ma non facile, data l'alta velocità cui normalmente essa opera.

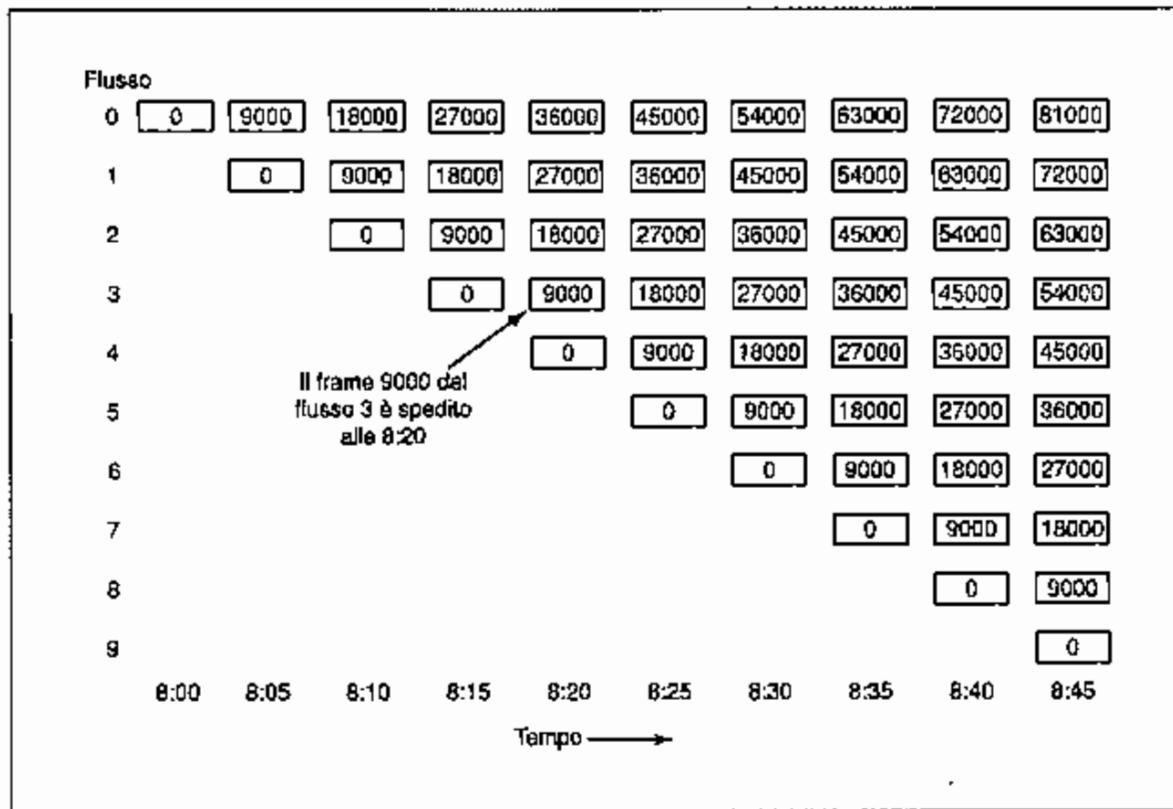
Tutto considerato, non esiste una soluzione facile. La sola strategia percorribile richiede una pianificazione preventiva: si può costruire un file speciale, contenente ad esempio un frame ogni dieci, e comprimerlo utilizzando il normale algoritmo MPEG; tale file è presentato nella Figura 7.3 come "avanti veloce". Per passare alla modalità avanti veloce, occorre che il server capisca a che punto si trovi l'utente nel file avanti veloce: ad esempio, se il frame corrente è il 48210, e il file avanti veloce è eseguito a velocità 10 x, il server deve localizzare il frame 4821 nel file avanti veloce, ed iniziare da quel punto l'esecuzione a velocità normale. Naturalmente, quel frame potrebbe essere P o B, ma il processo di decodifica dell'utente può saltare i frame, fino ad incontrarne uno di tipo I. L'esecuzione all'indietro è realizzata in modo analogo, utilizzando un secondo file speciale.

Quando l'utente ritorna a velocità normale, è necessario applicare alla rovescia lo stesso procedimento: se il frame corrente nel file avanti veloce è il 5734, il server passa al file regolare, e continua dal frame 57340. Di nuovo, se questo frame non è un frame I, il processo di decodifica dell'utente deve ignorare tutti i frame, sino ad incontrarne uno di tipo I.

Benché aggiungendo questi due file extra si risolva il problema, l'approccio presenta alcuni svantaggi. In primo luogo, occorre spazio aggiuntivo sul disco per memorizzare i file addizionali. Inoltre, è possibile eseguire l'avanti veloce ed il riavvolgimento soltanto a velocità corrispondente a quella dei file speciali. Infine, è necessaria ulteriore complessità per muoversi avanti e indietro tra i file regolare, avanti veloce e indietro veloce.

### 7.5.2 Video quasi su richiesta

Avere  $k$  utenti che scarichino lo stesso film, genera sostanzialmente lo stesso carico sul server di  $k$  utenti che scarichino  $k$  differenti film, ma un piccolo cambiamento nel modello rende possibili notevoli miglioramenti nelle prestazioni. Il problema del video su richiesta



**Figura 7.15** Il video quasi su richiesta ha un nuovo flusso che parte ad intervalli regolari, in questo esempio ogni cinque minuti [9000 frame].

è che gli utenti hanno la possibilità di iniziare a ricevere un film in un momento arbitrario; pertanto, se alle otto di sera cento utenti iniziano tutti a vedere un nuovo film, è assai probabile però che non lo facciano esattamente nel medesimo istante, e che non possano così condividere il flusso. Il cambiamento che rende possibile l'ottimizzazione, consiste nel comunicare agli utenti, ad esempio, che i film iniziano solo all'ora esatta, e poi ogni cinque minuti. Quindi, se un utente desidera vedere un film alle 8:02, dovrà aspettare fino alle 8:05.

Il vantaggio è che per un film di due ore sono necessari soltanto 24 flussi, senza relazione col numero degli utenti. La Figura 7.15 mostra come il primo flusso inizi alle 8; alle 8:05, quando il primo flusso è al frame 9000, inizia il secondo; alle 8:10 il primo flusso è al frame 18000 ed il secondo al 9000, ed inizia il terzo, e così sino al flusso 24, che parte alle 9:55. Alle 10 il primo flusso termina, e riparte con il frame 0. Questo modello è chiamato **video quasi su richiesta**, perché il video non parte proprio su richiesta, ma poco dopo.

Il parametro fondamentale è quanto spesso inizi un nuovo flusso. Se inizia ogni due minuti, saranno necessari 60 flussi per un film di due ore, ma il tempo massimo di attesa per l'utente sarà di due minuti. L'operatore deve decidere quanto a lungo il pubblico accetti di aspettare, perché quanto più lunga è l'attesa, più efficiente sarà il sistema, e più film potranno essere trasmessi insieme. Una strategia alternativa è avere un'opzione di attesa nulla, nel qual caso un nuovo flusso inizia all'istante, ma a un costo maggiore.

In un certo senso, il video su richiesta è come un taxi: si chiama, e arriva. Il video quasi su richiesta è come un autobus: ha un orario fisso e occorre attendere il successivo. Il trasporto di massa, tuttavia, ha senso solo se una massa esiste; nel centro di Milano, un bus che passi ogni cinque minuti raccoglierà almeno alcuni viaggiatori, mentre sulle strade secondarie dell'Appennino potrebbe essere spesso vuoto. Allo stesso modo, la tra-

smissione dell'ultimo film di Spielberg potrebbe attrarre abbastanza utenti da garantire un nuovo flusso ogni cinque minuti, mentre sarebbe meglio presentare *Via col vento* direttamente su richiesta.

Con il video quasi su richiesta gli utenti non hanno controlli VCR, e non possono mettere in pausa un film per farsi un giretto in cucina. L'unica possibilità, tornando dalla cucina, è passare al flusso partito più tardi, ripetendo quindi alcuni minuti di materiale.

Esiste anche un altro modello per il video quasi su richiesta. Invece di annunciare in anticipo agli spettatori che determinati film inizieranno ogni cinque minuti, si dà loro la possibilità di ordinarli quando vogliono. Ogni cinque minuti il sistema controlla le ordinazioni, e trasmette. Con questo approccio, un film può iniziare alle 8, 8 e 10, 8 e 15, 8 e 25, ma non in tempi intermedi in dipendenza della richiesta, e i flussi senza spettatori non sono trasmessi, salvando larghezza di banda del disco, memoria e capacità della rete. D'altro canto, passare dal congelatore è ora un po' a rischio, perché lo spettatore non ha garanzia che esista un altro flusso, cinque minuti dopo quello che stava seguendo. Naturalmente, l'operatore può fornire all'utente la possibilità di visualizzare una lista di tutti i flussi concorrenti, ma la maggioranza delle persone pensa che il proprio telecomando abbia già abbastanza bottoni, e probabilmente non sarebbe entusiasta di trovarne altri nuovi.

### 7.5.3 Video quasi su richiesta con funzioni VCR

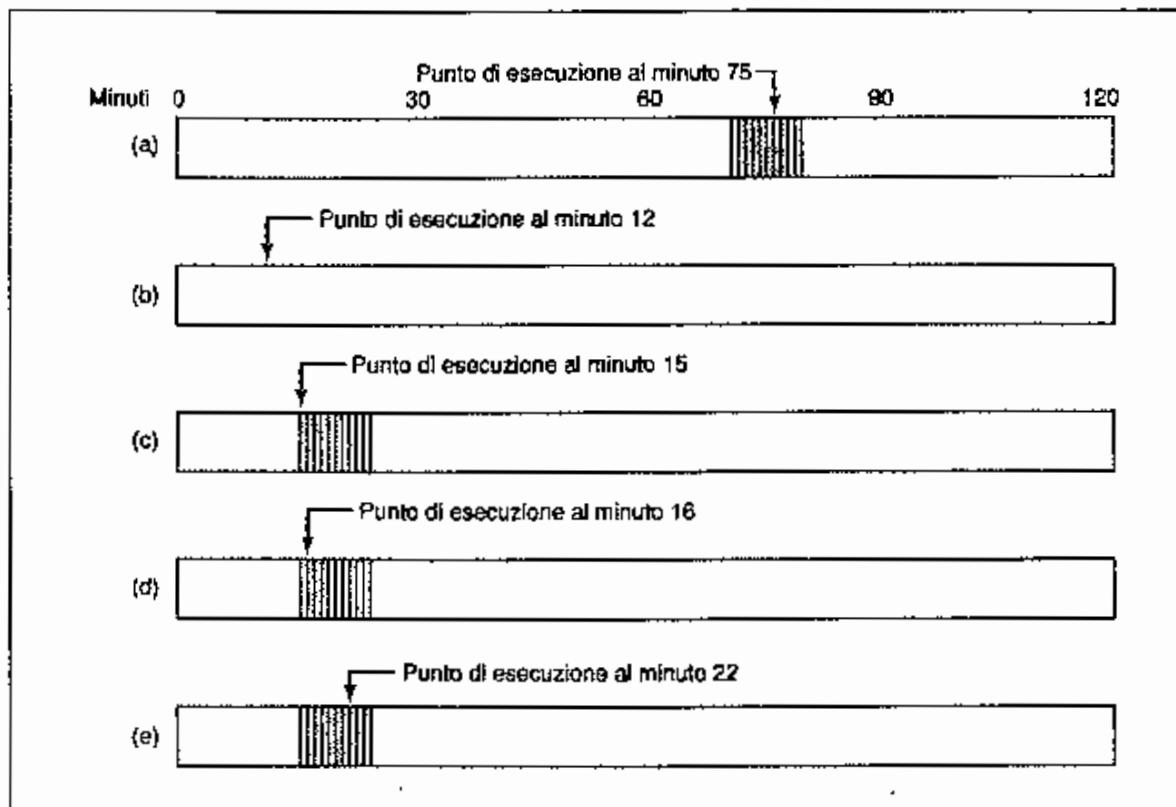
Sarebbe ideale una combinazione tra video quasi su richiesta (per l'efficienza) e controlli VCR per ogni spettatore (a vantaggio dell'utente); modificando un poco il modello, tale architettura è realizzabile. Di seguito, descriveremo in forma piuttosto semplificata un possibile procedimento per conseguire tale obiettivo (Abram-Profeta e Shin, 1998).

Iniziamo con lo schema del video quasi su richiesta della Figura 7.15. Comunque, aggiungiamo il vincolo che ciascuna macchina utente metta localmente nel buffer i precedenti  $\Delta T$  min e anche i successivi  $\Delta T$  min. Per i  $\Delta T$  min precedenti l'operazione è facile, in quanto basta salvarli dopo averli visualizzati. Per i  $\Delta T$  min successivi è più complessa, ma realizzabile se gli utenti possono leggere due flussi alla volta.

Un modo per preparare il buffer può essere illustrato con un esempio. Se un utente inizia a vedere un film alle 8:15, la macchina dell'utente legge e visualizza il flusso delle 8:15, che è al frame 0. In parallelo, legge e memorizza il flusso delle 8:10, che al momento è al quinto minuto, (cioè al frame 9000). Alle 8:20 i frame da zero a 17999 sono stati memorizzati, e l'utente sta aspettando di vedere i successivi 9000 frame. A quel punto il flusso delle 8:15 è scaricato, il buffer è riempito dal flusso delle 8:10 (che è al frame 18000), e la visualizzazione è ripresa dalla metà del buffer (frame 9000). Per ogni nuovo frame letto, ne viene aggiunto uno alla fine del buffer, e scaricato un altro all'inizio del buffer; il frame correntemente visualizzato, chiamato punto di esecuzione, è sempre alla metà del buffer. La Figura 7.16 (a) mostra la situazione dopo 75 minuti di film, con tutti i frame tra i minuti 70 e 80 contenuti nel buffer. Se la velocità di trasmissione dati è 4 Mbps, un buffer di dieci minuti richiede, per la sua memorizzazione, 300 milioni di byte. Ai prezzi correnti, il buffer può senz'altro essere tenuto su disco e possibilmente in RAM. Se si desidera utilizzare la RAM, ma 300 milioni di byte sono troppi, si può usare un buffer più piccolo.

Si supponga ora che l'utente decida di usare l'avanti veloce o l'indietro veloce. Quando il punto di esecuzione è compreso nell'intervallo tra 70 e 80 minuti, la visualizzazione può essere alimentata dal buffer, altrimenti se il punto di esecuzione si trova fuori da questo intervallo, siamo di fronte ad un problema, la cui soluzione consiste nel passare ad un canale privato (cioè un video su richiesta). Il movimento veloce in ciascuna direzione si può gestire con le tecniche precedentemente esaminate.

Normalmente, ad un certo momento l'utente vorrà di nuovo di vedere il film a velo-



**Figura 7.16** (a) Situazione iniziale. (b) Dopo un riavvolgimento fino al minuto 12. (c) Dopo un'attesa di 3 minuti. (d) Dopo aver iniziato a riempire il buffer. (e) Buffer pieno.

cità normale; a questo punto, si può pensare di spostare l'utente su uno dei canali del video quasi su richiesta, in modo da poter chiudere il canale privato. Si supponga, ad esempio, che l'utente decida di ritornare al minuto 12, come vediamo nella Figura 7.16 (b), che si trova fuori dal buffer, ed il dispositivo di visualizzazione non può quindi trarlo da esso. Inoltre, poiché il cambiamento è avvenuto istantaneamente al minuto 75, ci sono i canali che mostrano i film ai minuti 5, 10, 15, 20, ma nessuno al minuto 12.

La soluzione consiste nel continuare a visualizzare il canale privato, iniziando a riempire il buffer dal canale correntemente a 15 minuti nel film; dopo tre minuti, la situazione è come si presenta nella Figura 7.16. Il punto di esecuzione è ora 15 min, il buffer contiene i minuti da 15 a 18, ed i canali del video quasi su richiesta sono agli 8, 13, 18, 23 min: a questo punto è possibile scaricare il canale privato, ed alimentare il dispositivo di visualizzazione dal buffer. Il buffer continua ad essere riempito dal canale ora al minuto 18. Dopo un altro minuto, il punto di esecuzione è il minuto 16, il buffer contiene i minuti da 15 a 19, e il canale che lo riempie è al minuto 19, come nella Figura 7.16 (d).

Dopo altri sei minuti, il buffer è pieno, e il punto di esecuzione è al minuto 22, ma non è al centro del buffer; si può comunque eliminare questo problema, se necessario.

## 7.6 Posizionamento dei file

I file multimediali sono molto grandi, ma spesso sono scritti una volta sola, e letti molte volte; si tende ad accedere ad essi sequenzialmente, e la loro esecuzione deve inoltre soddisfare stretti criteri di qualità del servizio. L'insieme di queste esigenze suggerisce una stru-

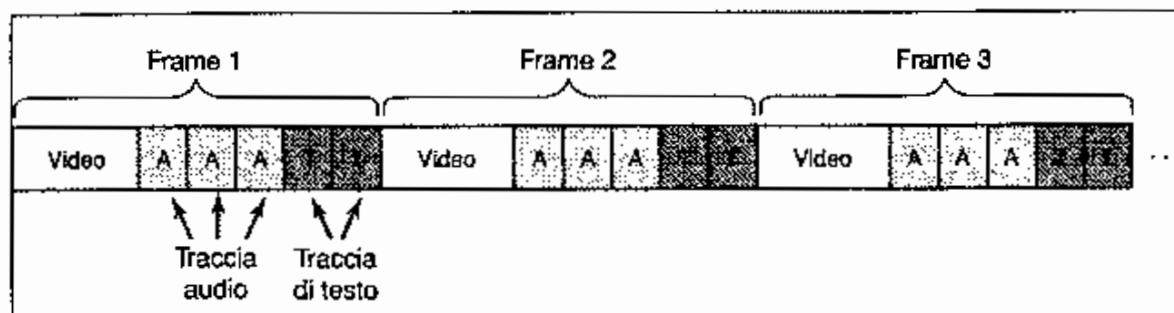


Figura 7.17 Interlacciamento di video, audio e testo in un singolo file contiguo per un film.

turazione diversa del file system rispetto a quella tradizionalmente usate dai sistemi operativi. Discuteremo alcuni di questi problemi, prima per un unico disco, poi per più dischi.

### 7.6.1 Posizionamento di un file su un unico disco

La prima esigenza da soddisfare è spedire i dati sulla rete o al dispositivo di uscita alla velocità richiesta e senza jitter; per questa ragione, aver più di una richiesta di posizionamento durante un frame è sicuramente non desiderabile. Un modo per eliminare richieste multiple di posizionamento all'interno dei file sui video server, consiste nell'utilizzare file contigui; di norma, i file contigui non funzionano bene, ma possono farlo su un video server attentamente precaricato in anticipo con film che non sono poi sostituiti.

Una complicazione tuttavia deriva dalla presenza di video, audio e testo, come vediamo nella Figura 7.3; anche se sono memorizzati come file separati e contigui, sarà necessario un posizionamento per andare dal file video a quello audio, e quindi al file di testo, se necessario. Questo suggerisce una seconda possibile strutturazione della memoria, con il video, l'audio e il testo interlacciato (Figura 7.17), ma con l'intero file ancora contiguo. Qui il video per il frame 1 è seguito dalle corrispondenti tracce audio, e quindi dalle tracce di testo sempre relative allo stesso frame. In dipendenza dal numero di tracce audio e testo, può essere più semplice leggere tutti i pezzi per ciascun frame in una singola operazione di lettura da disco, e trasinettere all'utente solo le parti necessarie.

Questa organizzazione richiede operazioni di I/O da disco in più, per leggere audio e testo non necessari, e spazio buffer di memoria in più per immagazzinarli, però elimina tutti i movimenti di ricerca (in un sistema ad utente singolo) e non richiede overhead per tener traccia della posizione dei frame sul disco, poiché l'intero film è in un unico file contiguo. L'accesso casuale con questa organizzazione non è possibile, ma, non essendo necessario, non è un problema. Allo stesso modo, avanti e indietro veloce sono impossibili senza strutture dati e complessità addizionali.

Il vantaggio di avere un film intero in un solo file contiguo si perde su un video server con canali di uscita multipli e concorrenti, perché, dopo aver letto un frame da un film, il disco dovrà leggere i frame di molti altri, prima di ritornare al film di partenza. Inoltre, per un sistema i cui film sono sia scritti, sia letti (ad esempio, un sistema per la produzione o l'elaborazione video), l'utilizzo di grandi file contigui è difficoltoso e ben poco utile.

### 7.6.2 Due strategie alternative per l'organizzazione dei file

Queste osservazioni conducono ad altre due organizzazioni di posizionamento per i file multimediali. La prima, il modello a blocco piccolo, è illustrata nella Figura 7.18 (a);

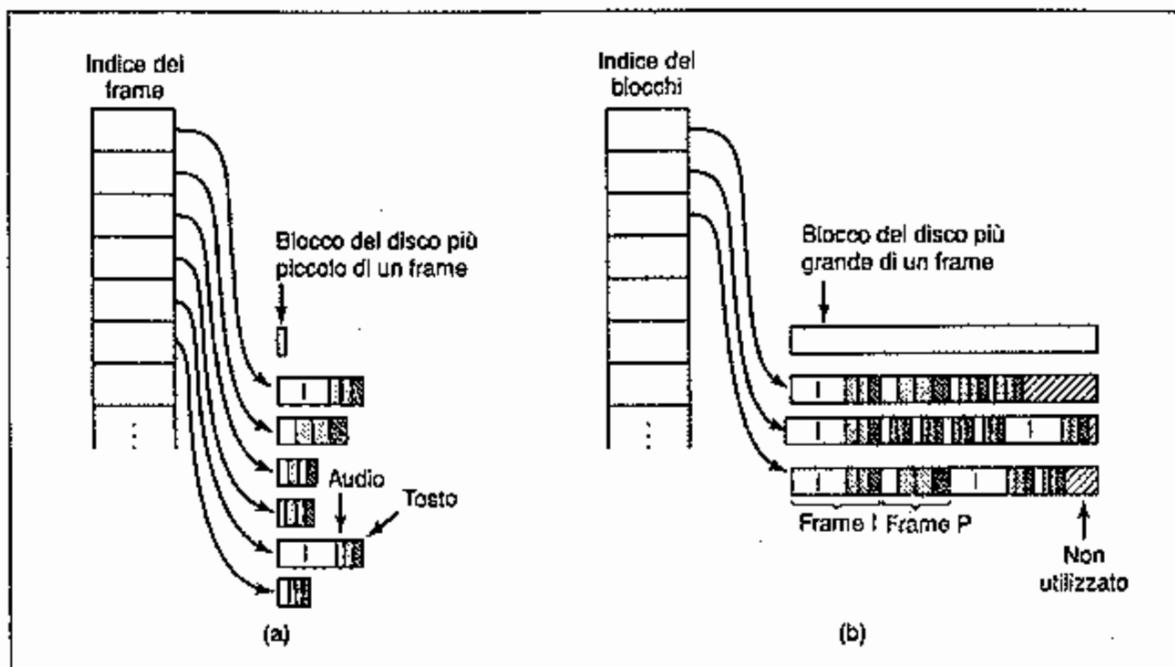


Figura 7.18 Memorizzazione non continua di film (a) Blocchi di disco piccoli. (b) Blocchi di disco grandi.

in questa organizzazione, si sceglie la dimensione del blocco del disco in modo che sia considerevolmente inferiore a quella media del frame, anche per i frame P e B. Per MPEG-2 a 4 Mbps, a 30 frame/s, il frame medio è 16 KB, cosicché un blocco di dimensione da 1 KB o 2 KB dovrebbe andare bene. L'idea qui è di avere una struttura dati, l'indice dei frame, per ciascun film, con una entry per ogni frame che punti all'inizio del frame stesso; ciascun frame contiene tutte le tracce video, audio e di testo per quel frame, memorizzate come insiemi contigui di blocchi di disco, come si vede nella figura. In tal modo la lettura del frame  $k$  richiede l'uso dell'indice dei frame per trovare la  $k$ -esima entry, e quindi leggere l'intero frame in una sola operazione. Poiché frame diversi hanno anche dimensioni diverse, la dimensione del frame (in blocchi) dev'essere contenuta nell'indice dei frame, ma anche con blocchi di 1 KB, un campo di 8-bit può gestire un frame sino a 255 KB, che è sufficiente per un frame NTSC non compresso, anche con molte tracce audio.

L'altro modo per memorizzare un film consiste nell'uso di un blocco di disco grande (ad esempio, 256 KB), e nel mettere diversi frame in ciascun blocco, come si vede nella Figura 7.18 (b). Un indice è ancora necessario, ma come indice dei blocchi, e non più dei frame; esso, infatti, è in sostanza lo stesso degli i-node della Figura 6.15, con l'eventuale aggiunta di informazione su quali frame si trovino all'inizio di ciascun blocco, per rendere possibile localizzare rapidamente un dato frame. In generale, un blocco non conterrà un numero intero di frame, e a ciò occorre in qualche modo provvedere; esistono due possibilità.

Nel primo caso, presentato in Figura 7.18 (b), quando il frame successivo non può essere contenuto nel blocco corrente, il resto del blocco è semplicemente lasciato vuoto. Lo spazio sprecato genera frammentazione interna, la stessa dei sistemi a memoria virtuale con pagine di dimensione fissa, ma non è mai necessario effettuare un movimento di ricerca all'interno del frame.

La seconda possibilità consiste nel riempire il blocco fino alla fine, suddividendo così i frame fra i blocchi; così è necessario il movimento di ricerca all'interno dei frame, che può ridurre le prestazioni, ma salva spazio sul disco, eliminando la frammentazione interna.

Per fare un confronto, l'uso di blocchi piccoli della Figura 7.18 (a) spreca anche spazio su disco, perché una frazione dell'ultimo blocco in ciascun frame non è utilizzata. Con

blocchi di dimensione di 1 KB, e un film NTSC di due ore, costituito da 216000 frame, lo spazio sprecato sarà di circa 108 KB su un totale di 3,6 GB; lo spazio sprecato è più difficile da calcolare per la Figura 7.18 (b), ma dovrà essere molto di più, perché di volta in volta ci saranno 100 KB lasciati liberi alla fine di un blocco, quando il frame successivo sarà un frame 1 maggiore di 100 KB.

D'altro canto l'indice dei blocchi è molto più piccolo di quello dei frame. Con un blocco di 256 KB e un frame medio di 16 KB, in un blocco sono contenuti circa 16 frame, cosicché un film con 216000 richiede solo 13500 entry nell'indice dei blocchi, contro le 216000 per l'indice dei frame. Per ragioni di prestazioni, in entrambi i casi l'indice dovrebbe elencare tutti i frame o i blocchi (in pratica non sono permessi blocchi indiretti come in UNDX): caricare 13500 entry da 8 byte in memoria (4 byte per l'indirizzo del disco, 1 byte per le dimensioni del frame e 3 per il numero del frame di inizio), contro 216000 entry da 5 byte (solo l'indirizzo sul disco e le dimensioni), salva almeno 1 MB di RAM mentre il film è in esecuzione.

Queste considerazioni portano ai seguenti compromessi:

1. Indice dei frame: utilizzo più pesante della RAM durante l'esecuzione del film; piccolo spreco di spazio su disco.
2. Indice dei blocchi (senza suddivisione dei frame fra blocchi): uso limitato della RAM; maggior spreco di spazio su disco.
3. Indice dei blocchi (con suddivisione dei frame fra blocchi): uso limitato della RAM, nessuno spreco di spazio su disco; movimenti di posizionamento extra.

I compromessi riguardano l'utilizzo della RAM durante l'esecuzione, lo spreco di spazio su disco, e le prestazioni inferiori durante l'esecuzione, causate dai necessari movimenti di posizionamento. È possibile tuttavia affrontare questi problemi in diversi modi: si può ridurre l'uso della RAM paginando parti della tabella dei frame. Il buffering può mascherare i movimenti di posizionamento durante la trasmissione dei frame, ma si rendono necessarie più memoria e più operazioni di copia. Un buon progetto deve analizzare tutti questi fattori e fare di volta in volta la scelta giusta per l'applicazione.

Inoltre, la gestione della memorizzazione sul disco è più complicata nella Figura 7.18 (a), poiché memorizzare un frame richiede di trovare un insieme consecutivo di blocchi della dimensione giusta, che idealmente non dovrebbe attraversare il limite di una traccia del disco (ma se si sfrutta lo skew della testina, non si perde gran che); in ogni caso, bisognerebbe evitare l'attraversamento del limite di un cilindro. Queste esigenze comportano che lo spazio libero del disco sia organizzato come una lista di buchi di dimensioni variabili, piuttosto che come una semplice lista di blocchi, o bitmap, ed entrambe possono essere usate nell'esempio relativo alla Figura 7.18 (b).

In tutti i casi, sono necessarie alcune osservazioni sul fatto di mettere tutti i blocchi o tutti i frame di un film entro uno stretto intervallo, ad esempio di pochi cilindri, ove possibile. Tale organizzazione significa che i movimenti di posizionamento andranno più veloci e rimarrà più tempo per altre attività (non in tempo reale), o per supportare canali video addizionali. Un posizionamento vincolato di questo tipo, si può ottenere dividendo il disco in gruppi di cilindri, e tenendo per ciascun gruppo liste o bitmap separate di blocchi liberi. Se ad esempio si impiegano i buchi, ci potrebbe essere una lista di buchi da 1 KB, una per buchi da 2 KB, una per quelli da 3 a 4 KB, e un'altra per buchi da 5 a 8 KB. Così, è facile trovare un buco di una certa dimensione, in un gruppo di cilindri.

Altra differenza tra i due approcci è la bufferizzazione: utilizzando i blocchi piccoli, ciascuna lettura ottiene esattamente un frame; di conseguenza, una semplice strategia di doppia bufferizzazione lavora bene; un buffer esegue il frame corrente, un altro carica il

successivo. Impiegando buffer di dimensione fissa, ciascuno dev'essere grande a sufficienza per contenere il frame I più grande possibile. Se si alloca da un pool un buffer diverso per ogni frame, e se la dimensione del frame è nota prima della lettura, si può scegliere un buffer più piccolo, per un frame P o B.

Per blocchi grandi, è necessaria una strategia più complessa, perché ciascun blocco contiene diversi frame, ed eventualmente frammenti di frame a ciascun estremo del blocco (secondo l'opzione scelta in precedenza). Se la visualizzazione e la trasmissione dei frame richiedono che essi siano contigui, occorre copiarli, ma la copia è un'operazione costosa, da evitare il più possibile; se la contiguità non è richiesta, i frame che si estendono ai confini dei blocchi si possono spedire in rete o al dispositivo di visualizzazione in due parti.

La doppia bufferizzazione si può anche usare con blocchi grandi, ma l'uso di due blocchi grandi comporta uno spreco di memoria, che si può diminuire impiegando un buffer di trasmissione circolare, un po' più ampio di un blocco del disco, per ogni canale, che alimenti la rete o il dispositivo di visualizzazione. Quando i contenuti del buffer scendono sotto di una certa soglia, il disco legge un nuovo blocco grande, il suo contenuto è copiato nel buffer di trasmissione e il buffer del blocco grande è restituito al pull comune. Occorre scegliere la dimensione del buffer circolare in modo tale che, quando si raggiunge il limite, ci sia ancora spazio per un altro blocco del disco. La lettura del disco non può andare direttamente al buffer di trasmissione perché potrebbe prodursi una sovrapposizione; si tratta di fare un compromesso tra la copiatura e l'uso della memoria.

Le prestazioni del disco costituiscono un altro elemento di confronto fra i due approcci: l'impiego di blocchi grandi consente al disco di raggiungere la massima velocità, caratteristica spesso di maggior importanza. Leggere piccoli frame P e B come unità separate non è conveniente. Inoltre, suddividere grandi blocchi su diversi drive (vedere in seguito), è possibile, mentre suddividere singoli frame su diversi drive non lo è.

L'organizzazione a piccoli blocchi della Figura 7.18 (a) è talvolta chiamata a lunghezza di tempo costante, perché ciascun puntatore nell'indice rappresenta lo stesso numero di millisecondi di esecuzione. Al contrario, l'organizzazione della Figura 7.18 (b) è definita a lunghezza di dati costante, perché i blocchi dei dati sono della stessa dimensione.

Un'altra differenza fra le due organizzazioni di file è che se i tipi dei frame sono memorizzati nell'indice della Figura 7.18 (a), è possibile effettuare un avanti veloce semplicemente visualizzando i frame I. Comunque, secondo quanto spesso i frame I appaiono nel flusso, la velocità può essere percepita come troppo veloce o troppo lenta. In ogni caso, con l'organizzazione della Figura 7.18 (b), l'avanti veloce non è possibile. In effetti, leggere il file sequenzialmente per raggiungere i frame desiderati richiede un massiccio I/O da disco.

È anche possibile servirsi di un file speciale che, eseguito a velocità normale, dia l'illusione di avanti veloce a velocità 10 x; si può strutturare questo file come gli altri, servendosi di un indice dei frame o di un indice dei blocchi. Quando si apre un file il sistema dev'essere in grado di trovare il file avanti veloce; poi, quando serve (cioè quando l'utente preme il pulsante avanti veloce), il sistema deve aprirlo all'istante, e quindi saltare al posto corretto nel file. In questo caso il sistema conosce il numero del frame a cui si trova correntemente, e deve poter localizzare il frame corrispondente nel file avanti veloce: se ci si trova al frame 4816 e si sa che il file avanti veloce è 10 x, allora occorre localizzare il frame 482 in quel file ed iniziare da lì l'esecuzione.

Se si usa un indice dei frame, localizzarne uno specifico è facile: basta usare l'indice dei frame; usando invece l'indice dei blocchi, occorre informazione extra in ciascuna entry per identificare quali frame siano in un certo blocco, ed occorre eseguire una ricerca binaria sull'indice dei blocchi. Indietro veloce funziona in modo analogo.

### 7.6.3 Posizionare i file per i video quasi su richiesta

Abbiamo esaminato le strategie inerenti i video su richiesta; per i video quasi su richiesta è più efficiente adottare una strategia diversa di posizionamento dei file, sapendo che lo stesso film è eseguito come un insieme di diversi flussi sfalsati l'uno rispetto all'altro, ed anche se il film è memorizzato come un file contiguo, è necessario il movimento di posizionamento per ciascun flusso. Chen e Thapar (1997) hanno proposto una strategia per eliminare quasi tutti i movimenti di posizionamento, illustrata nella Figura 7.19 per un film eseguito a 30 frame/s, con un nuovo flusso ogni 5 minuti, come in Figura 7.15. Con questi parametri sono necessari 24 flussi concorrenti per un film di due ore.

In questo posizionamento, insiemi di 24 frame sono concatenati e scritti su disco come un singolo record, ed è possibile leggerli anche all'indietro con una sola operazione di lettura. Si consideri l'istante in cui il flusso 24 sta per partire: avrà bisogno del frame 0. Il flusso 23 che è partito 5 minuti prima, avrà bisogno del frame 9000, il flusso 22 avrà bisogno del frame 18000, e così via all'indietro fino al flusso 0, che avrà bisogno del frame 207000. Mettendo questi frame consecutivamente su una traccia del disco, il video server può soddisfare tutti e 24 i flussi in ordine inverso con un solo movimento di posizionamento (al frame 0). Naturalmente, è possibile riversare su disco i frame, nel caso in cui si debbano servire i flussi in ordine ascendente. Dopo che l'ultimo flusso è stato servito, il braccio del disco può muoversi sulla traccia 2 per prepararsi di nuovo a servirli tutti. Questo schema non richiede che l'intero file sia contiguo, ma con diversi flussi eseguiti simultaneamente, dà buone prestazioni.

Una strategia semplice è quella della doppia bufferizzazione: mentre un buffer viene scaricato su 24 flussi, si carica in anticipo un altro, e quando quello corrente è finito, i due buffer vengono scambiati e quello appena usato è caricato tramite una singola operazione del disco.

Un problema interessante è quello del dimensionamento del buffer. Esso deve contenere 24 frame, ma, poiché le dimensioni dei frame sono variabili, non è del tutto banale scegliere la corretta dimensione del buffer. Rendere il buffer di dimensione adeguata a 24 frame è eccessivo, mentre renderlo sufficiente soltanto per 24 frame medi è rischioso.

Fortunatamente, per ogni determinato film la traccia più grande (nel senso della Figura 7.19) è nota in anticipo, ed è possibile scegliere un buffer di dimensioni adeguate; ma

| Ordine in cui si leggono i blocchi da disco → |              |              |       |       |       |       |       |       |       |       |              |             |
|-----------------------------------------------|--------------|--------------|-------|-------|-------|-------|-------|-------|-------|-------|--------------|-------------|
|                                               | Flusso<br>24 | Flusso<br>23 |       |       |       |       |       |       |       |       | Flusso<br>15 | Flusso<br>1 |
| Traccia 1                                     | 0            | 9000         | 18000 | 27000 | 36000 | 45000 | 54000 | 63000 | 72000 | 81000 | ...          | 207000      |
| Traccia 2                                     | 1            | 9001         | 18001 | 27001 | 36001 | 45001 | 54001 | 63001 | 72001 | 81001 | ...          | 207001      |
| Traccia 3                                     | 2            | 9002         | 18002 | 27002 | 36002 | 45002 | 54002 | 63002 | 72002 | 81002 | ...          | 207002      |

Frame 27002 (circa 15 min dall'inizio del film)

Figura 7.19 Posizionamento ottimale dei frame per i video quasi su richiesta.

potrebbe accadere che nella traccia più grande ci siano ad esempio 16 frame I, mentre nella seconda come grandezza ci siano solo 9 frame I: scegliere un buffer ampio abbastanza per questa seconda traccia, potrebbe essere più saggio. Ciò impone di troncare la traccia più grande, cioè sottrarre ad alcuni flussi un frame del film; per evitare uno sfarfallio, si può rivisualizzare il frame precedente: non se ne accorgerà nessuno.

Proseguendo con questo approccio, se la terza traccia in ordine di grandezza ha solo 4 frame I, utilizzare un buffer sufficiente per 4 frame I e 20 frame P, può essere sensato: introdurre per due volte due frame ripetuti per qualche flusso, è probabilmente accettabile. Quando finisce tutto il ragionamento? Probabilmente quando la dimensione del buffer è abbastanza grande per il 99% dei frame: chiaramente c'è un compromesso fra l'uso della memoria e la qualità dei film visualizzati. Si noti che più flussi simultanei ci saranno, migliori saranno le statistiche e più uniformi gli insiemi di frame.

## 7.6.4 Posizionare più file su un unico disco

In precedenza abbiamo considerato il posizionamento di un singolo film; naturalmente su un video server ci saranno molti film, e se sono disseminati casualmente sul disco si perderà del tempo nello spostare la testina del disco da film a film, quando film diversi sono visti simultaneamente da utenti diversi.

È possibile migliorare questa situazione considerando che alcuni film sono più seguiti di altri, e tenendo conto di questo nel posizionarli sul disco. Benché non si abbiano elementi certi per prevedere la popolarità di alcuni film (tranne osservare che la presenza di star famose tende ad incrementarla) si può osservare qualcosa sulla popolarità relativa dei film.

Per molti indicatori di popolarità, come il noleggio di un film, la consultazione di un libro in biblioteca, il numero di contatti ad una pagina WEB, la ricorrenza di una parola in un romanzo, e l'incremento demografico delle più grandi città, un'approssimazione ragionevole della popolarità relativa segue uno schema sorprendentemente prevedibile. Questo schema è stato scoperto a Harvard da un professore di linguistica, George Zipf (1902/1950), ed è ora chiamato legge di Zipf. La legge afferma che se i film, i libri, le pagine Web o le parole sono ordinate secondo la loro popolarità, la probabilità che l'utente successivo scelga l'oggetto in posizione  $k$  nella lista è  $C/k$ , dove  $C$  è una costante di normalizzazione.

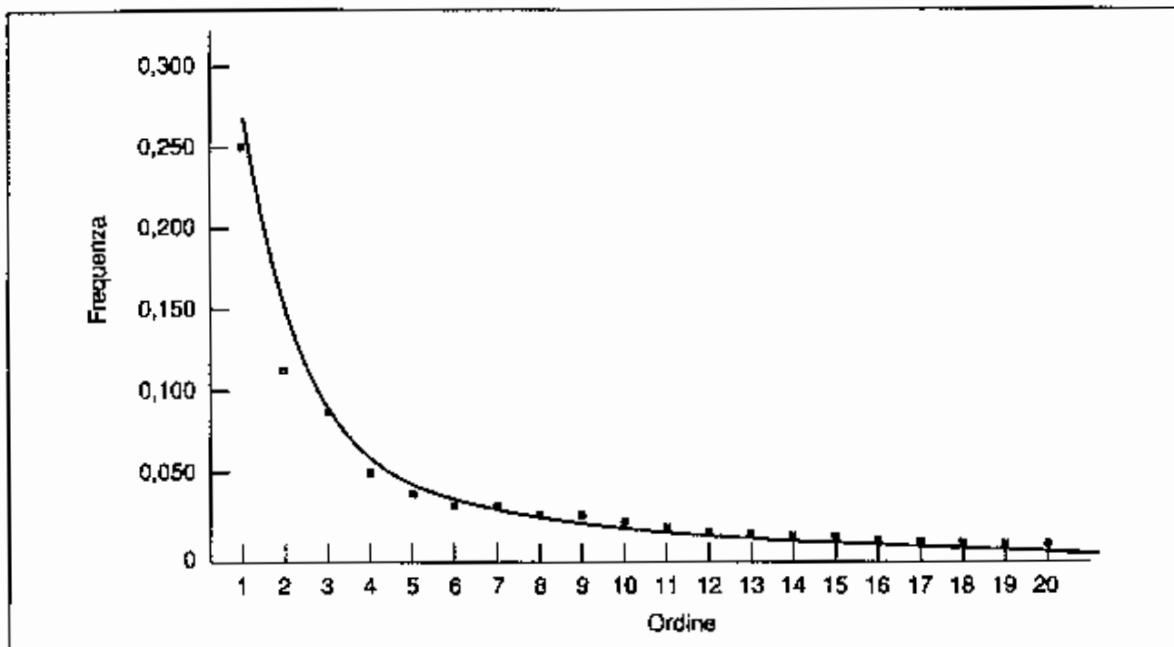
Pertanto, la frazione di scelte per i primi tre film sarà rispettivamente  $C/1$ ,  $C/2$  e  $C/3$ , e la somma di tutte le frazioni deve dare 1. In altre parole, se ci sono  $N$  film, allora

$$C/1 + C/2 + C/3 + C/4 + \dots + C/N = 1$$

Da questa equazione si può calcolare  $C$ : i valori di  $C$  per popolazioni con 10, 100, 1000 e 10000 elementi sono rispettivamente 0,341, 0,193, 0,134 e 0,102; per esempio, su mille film, le probabilità per i primi cinque sono 0,134, 0,067, 0,045, 0,034 e 0,027.

La legge di Zipf è illustrata nella Figura 7.20: per divertimento, è stata applicata alle venti maggiori città degli USA. La legge di Zipf predice che la seconda di esse dovrebbe avere una popolazione pari alla metà della prima, la terza pari ad un terzo, e così via; benché tutt'altro che perfetta, fornisce un'approssimazione sorprendentemente buona.

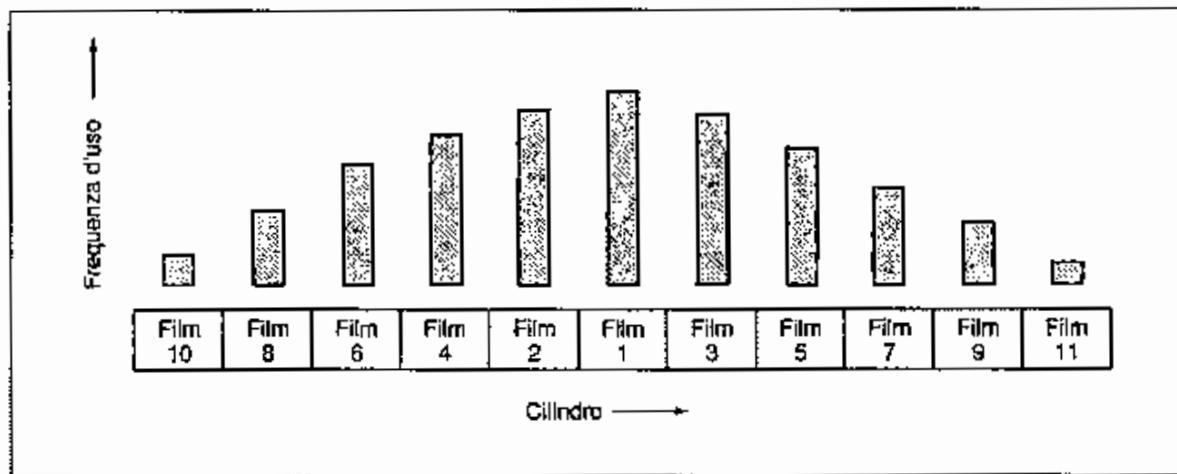
Per i film su un video server, la legge di Zipf afferma che il film più popolare è scelto il doppio delle volte rispetto al secondo, tre volte rispetto al terzo e così via. Sebbene la distribuzione all'inizio diminuisca rapidamente, essa mostra una lunga coda. Ad esempio, il film 50 ha una popolarità di  $C/50$  ed il film 51 ne ha una di  $C/51$ , di modo che il film 51 risulta popolare 50/51 rispetto al film 50, con una differenza di solo il 2%. Man-



**Figura 7.20** La curva descrive la legge di Zipf per  $N = 20$ . I quadrati rappresentano la popolazione delle venti maggiori città degli USA ordinate per numero di abitanti (New York è 1, Los Angeles è 2, Chicago è 3 e così via).

mano che si prosegue sulla coda, la differenza percentuale fra due film consecutivi diviene sempre minore; è dunque possibile concludere che il server ha bisogno di molti film perché c'è molta richiesta per i film dopo i primi dieci.

La conoscenza delle popolarità relative dei vari film rende possibile costruire un modello per le prestazioni di un video server, ed usare quelle informazioni per posizionare i file. Diversi studi hanno mostrato che la migliore strategia è sorprendentemente semplice e indipendente dalla distribuzione. È nota come **algoritmo a canne d'organo** (Grossman e Silverman, 1973; Wong, 1983), e consiste nel posizionare il film più popolare a metà del disco, con il secondo ed il terzo ai suoi lati. All'esterno di questi, vengono i numeri 4 e 5 e così via (Figura 7.21). Tale posizionamento funziona al meglio se ciascun film è un file contiguo, del tipo presentato nella Figura 7.17, ma può anche essere usato,



**Figura 7.21** Distribuzione a canne d'organo dei file di un video server.

entro certi limiti, se ciascun film è vincolato in uno stretto intervallo di cilindri. Il nome dell'algoritmo deriva dal fatto che un istogramma delle probabilità appare come un organo piuttosto asimmetrico.

Il compito di questo algoritmo è cercare di tenere la testina a metà del disco. Con mille film e una distribuzione in base alla legge di Zipf, i primi cinque film corrispondono ad una probabilità totale di 0,307, il che significa che la testina del disco rimarrà nei cilindri allocati per i primi cinque film per il 30% circa del tempo, una percentuale molto grande, avendo a disposizione mille film.

### 7.6.5 Posizionamento dei file su dischi diversi

Per ottenere migliori prestazioni, spesso i video server hanno molti dischi che funzionano in parallelo. Talvolta, si utilizzano i RAID, ma spesso non perché offrono una maggiore affidabilità, a spese delle prestazioni: i video server di norma richiedono alte prestazioni, e non si curano troppo della correzione di errori occasionali. Anche i controller dei RAID possono diventare un collo di bottiglia, se hanno troppi dischi da gestire contemporaneamente.

Una configurazione più comune consiste in un gran numero di dischi, talvolta chiamati *disk farm* (alla lettera, fattoria di dischi), che non ruotano in modo sincronizzato e non contengono bit di parità come i RAID. Un'altra possibilità consiste nel mettere il film A sul disco 1, il film B sul 2 e così via, come vediamo nella Figura 7.22 (a). In pratica, con i dischi moderni, diversi film possono essere contenuti nello stesso disco.

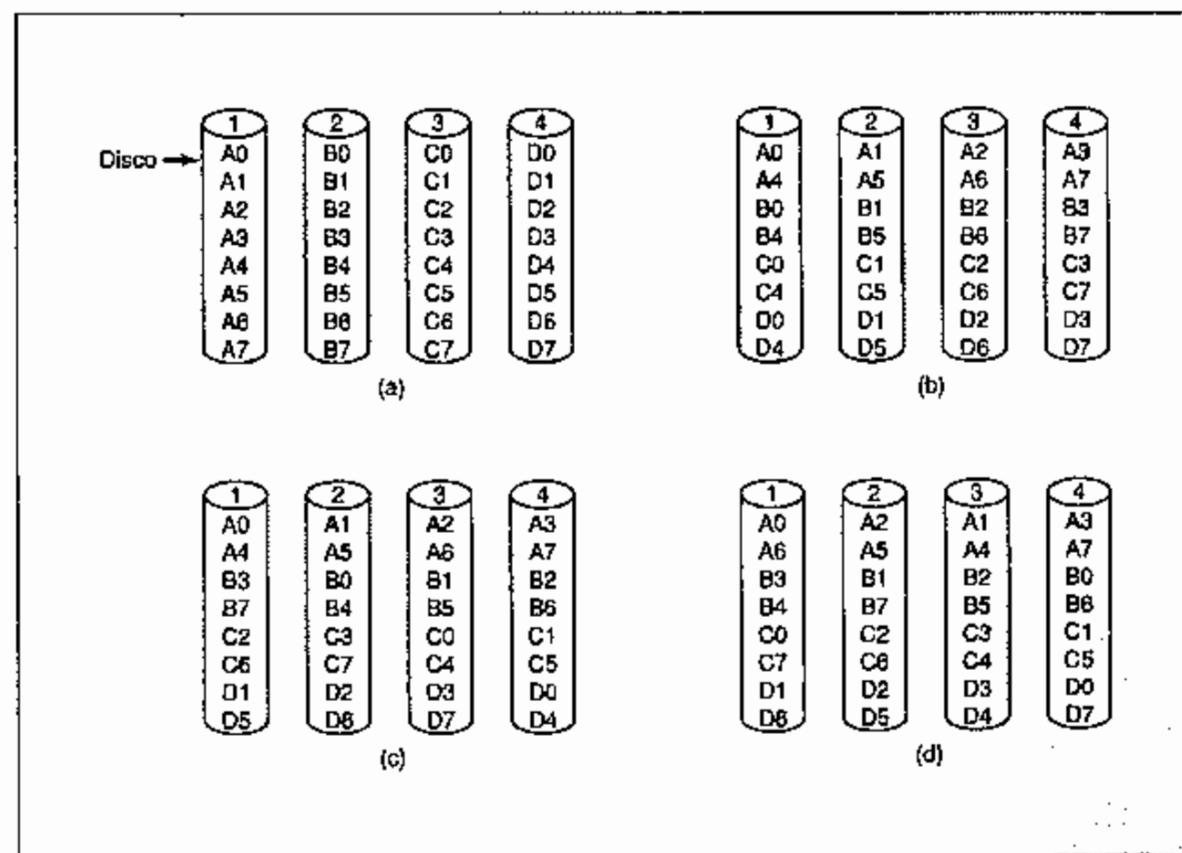


Figura 7.22 Quattro modi di organizzare i file multimediali su dischi diversi. (a) Nessuna suddivisione. (b) Lo stesso schema di suddivisione per tutti i file. (c) Suddivisione sfalsata. (d) Suddivisione casuale.

Quest'organizzazione è semplice da implementare, e ha semplici caratteristiche in caso di rottura: se un disco si rompe, tutti i film su di esso diventano inutilizzabili. Notare che se un'azienda perde un disco pieno di film non subisce un danno comparabile alla perdita di un disco pieno di dati: i film si possono ricaricare facilmente da un DVD, su un disco di scorta. Uno svantaggio di questo approccio è il caricamento eventualmente non ben bilanciato: se alcuni dischi contengono film molto richiesti, e altri invece solo film meno popolari, il sistema non sarà pienamente utilizzato. Una volta note le frequenze di utilizzo dei film, è ovviamente possibile spostarne alcuni per bilanciare a mano il carico.

Una seconda possibile organizzazione è suddividere uno stesso film su diversi dischi, quattro nell'esempio della Figura 7.22 (b). Si supponga per ora che tutti i frame siano della stessa dimensione (cioè non compressi): scriviamo un numero fissato di byte del film *A* sul disco 1, quindi lo stesso numero di byte sul 2, e così via fino a raggiungere l'ultimo disco, in questo caso con l'unità A3. In seguito, la suddivisione continua ancora sul primo disco con A4, e così via, finché l'intero file è stato scritto. A quel punto, i film *B*, *C* e *D* sono suddivisi seguendo lo stesso schema.

Un possibile svantaggio di tale schema è che, iniziando tutti i film sul primo disco, il carico non sia bilanciato su diversi dischi; una migliore distribuzione del carico consiste nello sfalsare i dischi di avvio, come si vede nella Figura 7.22 (c). È possibile inoltre suddividere il carico utilizzando uno schema di successione casuale per ciascun file, come nella Figura 7.22 (d).

Abbiamo appena ipotizzato che tutti i frame siano della stessa dimensione, ma con i film MPEG-2 questo non accade: i frame I sono molto più grandi dei frame P. Ci sono due modi di trattare questo problema, e cioè suddividere per frame, oppure per blocchi: nel primo caso, il primo frame del film *A* va sul disco 1 come un'unità contigua, indipendentemente dalle sue dimensioni; il frame successivo va sul disco 2, e così via. In modo simile si suddivide il film *B*, o partendo dallo stesso disco o da quello successivo (se sfalsato), oppure da un disco scelto a caso. Poiché i frame sono letti uno per volta, questa forma di suddivisione non accelera la lettura di un film dato, ma suddivide il carico sui dischi molto meglio di quanto rappresentato nella Figura 7.22 (a), dove si possono avere pessime prestazioni, se molti decidono di vedere il film *A* la stessa notte, e nessuno sceglie il film *C*. Nel complesso, suddividendo il carico su tutti i dischi, fa un uso migliore della larghezza di banda totale del disco, e quindi si incrementa il numero di clienti serviti.

Un altro modo di suddividere è quello per blocchi: per ciascun film, si scrivono in successione (o casualmente) unità di dimensione fissa, e ciascun blocco contiene uno o più frame, o frammenti di esso. Il sistema può ora farsi carico di richieste per blocchi diversi, nello stesso momento per lo stesso film; ciascuna richiesta legge i dati in un diverso blocco di memoria, ma in modo tale che, completate tutte le richieste, una parte contigua del film (contenente molti frame), sia ora assemblata in memoria in modo contiguo. Queste richieste possono procedere in parallelo, e dopo aver soddisfatto l'ultima, si può segnalare al processo richiedente il completamento del lavoro, ed esso inizia a trasmettere i dati all'utente. Dopo un certo numero di frame, quando il buffer è arrivato agli ultimi frame, partono altre richieste per precaricare un altro buffer. Questo approccio usa grandi quantità di memoria per la bufferizzazione, al fine di mantenere i dischi occupati: in un sistema con mille utenti attivi e un buffer di 1 MB (ad esempio con blocchi di 256 KB in ciascuno dei quattro dischi), è necessario 1 GB di RAM per i buffer. Tale quantità è poca cosa per un server da mille utenti, e non dovrebbe costituire un problema.

L'ultimo problema inherente alla suddivisione è stabilire quanti dischi usare per la suddivisione stessa: in un caso estremo, ciascun film è suddiviso su tutti i dischi. Ad esempio, con film di 2 GB e mille dischi, potrebbe essere scritto un blocco di 2 MB su ciascun disco, in modo che nessun film utilizzi lo stesso disco due volte. Nel caso opposto, i dischi sono partizionati in piccoli gruppi (Figura 7.22) e ciascun film è ristretto in una singola

partizione. La prima, chiamata **suddivisione ampia**, effettua un buon lavoro di bilanciamento del carico sui dischi, ma lascia un problema notevole: poiché ogni film usa ogni disco, se un disco va fuori uso, non è possibile visualizzare nessun film. La seconda, chiamata **suddivisione stretta**, può subire gli effetti di punti critici (partizioni molto richieste), ma la perdita di un disco rovina solo i film della sua partizione. La suddivisione di frame di dimensione variabile è analizzata matematicamente in dettaglio in (Shenoy e Vin, 1999).

## 7.7 Meccanismi di cache

I tradizionali meccanismi LRU di cache dei file, non funzionano bene con i file multimediali perché gli schemi di accesso per i film sono diversi da quelli dei file di testo. L'idea alla base dei tradizionali buffer cache LRU è che dopo l'utilizzo di un blocco, esso dovrebbe essere mantenuto nella cache, in caso sia necessario riutilizzarlo di nuovo in modo rapido. Ad esempio, quando si elabora un file, l'insieme dei blocchi su cui è scritto tende ad essere usato più volte, sino alla fine della sessione di elaborazione; in altre parole, quando esiste una probabilità relativamente elevata di riutilizzare un blocco entro breve tempo, è opportuno mantenerlo per eliminare futuri accessi ad esso.

Con i multimedia, lo schema di accesso usuale è che un film sia visto sequenzialmente dall'inizio alla fine: è improbabile che un blocco sia utilizzato una seconda volta, sempre che l'utente non riavvolga il film per rivedere alcune scene. Di conseguenza le normali tecniche di cache non funzionano; tuttavia, la cache può ancora essere d'aiuto se usata in modo diverso. Nelle prossime sezioni, prenderemo in esame la cache per i multimedia.

### 7.7.1 Meccanismi di cache per i blocchi

Benché sia inutile mantenere un blocco in memoria, sperando di usarlo di nuovo entro breve tempo, la prevedibilità dei sistemi multimediali si può sfruttare per rendere nuovamente utili i meccanismi di cache. Si supponga che due utenti stiano guardando lo stesso film, e che uno di loro abbia iniziato due secondi dopo l'altro; dopo che il primo utente ha prelevato e visualizzato un dato blocco, è molto probabile che l'altro abbia bisogno dello stesso blocco due secondi più tardi. Il sistema può facilmente mantenere traccia di quali film abbiano un solo spettatore e di quali ne abbiano due o più, molto vicini nel tempo.

Quindi, se è letto un blocco di un film che sarà di nuovo necessario entro breve tempo, può essere sensato metterlo nella cache, a seconda di quanto a lungo debba essere mantenuto nella cache stessa, e di quando limitata sia la memoria. Invece di mantenere tutti i blocchi del disco nella cache, e scaricare quello usato meno di recente quando la cache è piena, si dovrebbe usare una diversa strategia. Ogni film che abbia un secondo spettatore che segue il primo ad un intervallo di tempo  $\Delta t$ , può essere marcato per la cache e tutti i suoi blocchi potranno essere posti in essa finché il secondo spettatore (e probabilmente un terzo) li avranno utilizzati. Per gli altri film, non si usa alcun meccanismo di cache.

Questa idea può essere sviluppata: talvolta, è ammissibile effettuare il merge di due flussi. Si supponga che due utenti stiano guardando lo stesso film a dieci secondi di distanza l'uno dall'altro: tenere i blocchi nella cache per dieci secondi, è possibile, ma spreca memoria. Un approccio alternativo, basato su un trucco, consiste nel provare a mettere i due film in sincronia: questo si ottiene cambiando la velocità dei frame per entrambi i film, come si vede nella Figura 7.23.

Nella Figura 7.23 (a), entrambi i film vanno alla velocità standard NTSC di

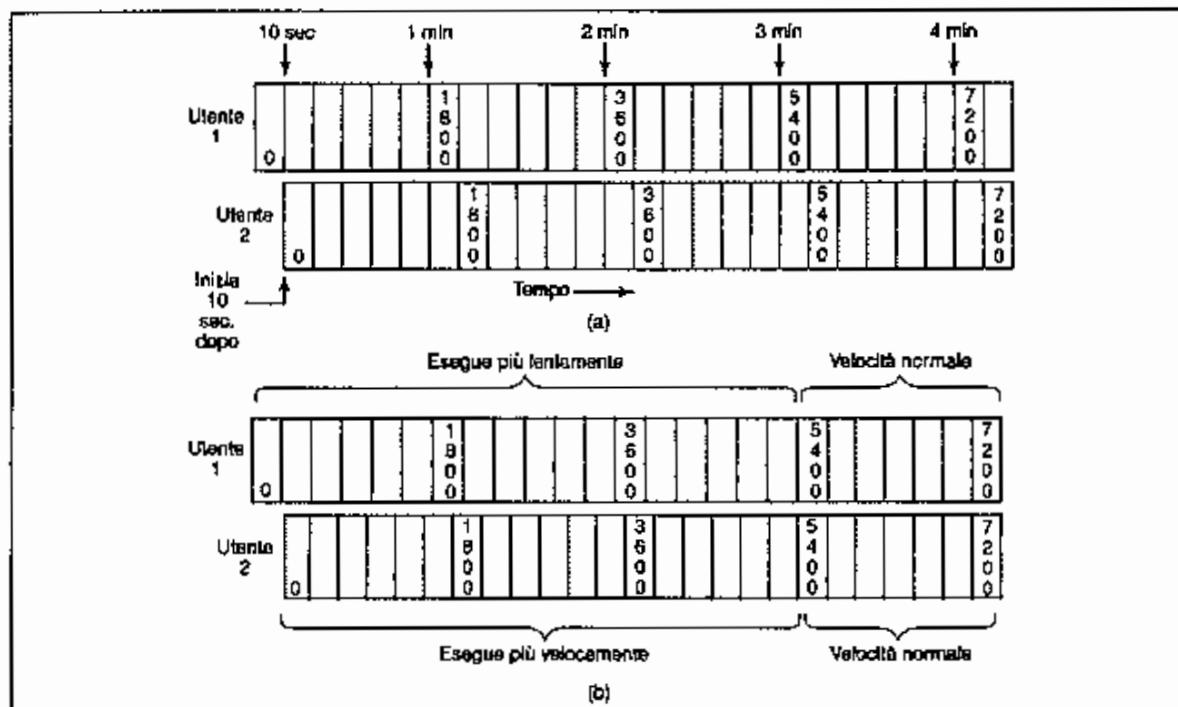


Figura 7.23 (a) Due utenti guardano lo stesso film sfalsato di dieci secondi. (b) Effettuare il merge di due flussi.

1800 frame/min, e poiché l'utente 2 ha iniziato dieci secondi più tardi, conserverà questo ritardo per l'intero film. Nella Figura 7.23 (b), il flusso dell'utente 1 viene rallentato quando appare l'utente 2: invece di andare a 1800 frame/min, va a 1750 frame/min per i successivi tre minuti. Dopo tre minuti, si trova al frame 5550. Inoltre, il flusso dell'utente 2 viene eseguito a 1850 frame/min per i primi tre minuti, dopodiché si troverà anch'esso al frame 5550; da qui in poi, entrambi i film sono eseguiti a velocità normale.

Nell'arco di tempo in cui si sincronizzano, il flusso dell'utente 1 va più lento del 2,8%, e quello dell'utente 2 va più veloce del 2,8%. È improbabile che gli utenti se ne accorgano, ma, se ciò accadesse, il periodo di sincronizzazione si potrebbe estendere oltre i tre minuti.

Una possibilità per rallentare il flusso di un utente al fine di effettuare il merge con un altro flusso, consiste nell'offrire l'opzione di avere della pubblicità nei propri film, in cambio di un prezzo minore rispetto ai film senza pubblicità. L'utente può anche scegliere le categorie di prodotti, cosicché la pubblicità sarà meno intrusiva, e probabilmente più seguita. Manipolando il numero, la lunghezza e la temporizzazione degli spot, si può fermare il flusso per un tempo sufficiente a sincronizzarlo con un flusso dato (Krishnan 1999).

## 7.7.2 Meccanismi di cache dei file

I meccanismi di cache possono essere utili nei sistemi multimediali anche in un altro modo: per le grandi dimensioni di quasi tutti i film (2GB), i video server spesso non possono memorizzare tutti i loro film sul disco, e li conservano su DVD o su nastro. Quando c'è bisogno di un film, lo si può sempre copiare sul disco, ma occorre un certo tempo di avviamento per localizzare il film e copiarlo sul disco; perciò la maggioranza dei video server mantiene una cache del disco per i film richiesti più spesso, con i film popolari memorizzati tutti su disco.

Un altro modo per servirsi dei meccanismi di cache consiste nel mantenere i primi

minuti di ciascun film su disco: quando un film è richiesto, l'esecuzione parte immediatamente dal file del disco; nel frattempo, si copia il film sul disco, da un nastro o da DVD. Tenendo sempre in memoria una parte sufficiente del film, è molto probabile che la parte successiva del film sarà estratta prima che ce ne sia bisogno. Se tutto va bene, il film intero sarà disponibile sul disco ben prima del necessario. Quindi, andrà nella cache e rimarrà sul disco nel caso ci siano successive richieste. Se passa troppo tempo prima della nuova richiesta, il film sarà rimosso dalla cache per far spazio ad un altro più seguito.

## 7.8 Schedulazione del disco per i multimedia

I multimedia pongono ai dischi richieste diverse, rispetto alle applicazioni tradizionali orientate al testo, ad esempio i compilatori o i programmi di scrittura; in particolare, i multimedia richiedono una velocità di trasmissione dati assai elevata ed una spedizione dei dati in tempo reale. Non è banale provvedere a queste richieste; inoltre, nel caso di un video server, c'è una pressione economica per avere un singolo server che gestisca simultaneamente migliaia di utenti, e tali esigenze hanno un impatto sull'intero sistema. In precedenza, abbiamo esaminato il file system; passiamo ora alla schedulazione del disco per i multimedia.

### 7.8.1 Schedulazione statica del disco

Benché i multimedia pongano richieste onerose, in termini di tempo reale e velocità di trasmissione, a tutte le parti del sistema, sono anche dotati di una proprietà che li rende più semplici da gestire rispetto ad un sistema tradizionale: la prevedibilità. In un sistema operativo tradizionale, le richieste per blocchi del disco sono effettuate in un modo abbastanza imprevedibile; il meglio che il sottosistema del disco può fare è effettuare la lettura di un blocco in anticipo per ciascun file aperto; oltre a ciò, può solo aspettare che arrivino richieste e quindi elaborarle. Il multimedia è diverso: ciascun flusso attivo pone un carico sul sistema ben definito e altamente prevedibile. Per l'esecuzione NTSC, ogni 33,3 ms ciascun utente vuole il frame successivo del suo file, e il sistema dispone di 33,3 ms per fornire tutti i frame (il sistema ha bisogno di bufferizzare almeno un frame per flusso, in modo che il prelievo del frame  $k + 1$  possa procedere in parallelo con l'esecuzione del frame  $k$ ).

Questo carico prevedibile può essere utilizzato per schedulare il disco con algoritmi appositi per le operazioni multimediali; di seguito, considereremo un solo disco, ma l'idea può essere applicata anche a più dischi. In questo esempio, ipotizzeremo che ci siano dieci utenti, ciascuno dei quali guarda un film diverso, e che tutti i film abbiano la stessa risoluzione, velocità dei frame e altre proprietà.

In dipendenza dal resto del sistema, il computer può avere dieci processi, uno per ogni flusso video, un processo con dieci thread, oppure un processo con un thread che gestisce dieci flussi in modo round-robin. I dettagli non sono importanti, l'essenziale è la suddivisione del tempo in round, dove un round è il tempo di un frame (33,3 ms per NTSC, 40 ms per PAL). All'inizio di ciascun round, si genera una richiesta del disco per utente, come vediamo nella Figura 7.24.

Dopo che all'inizio del round sono arrivate tutte le richieste, il disco sa cosa deve fare durante quel round, e sa che non arriveranno altre richieste, finché le prime non siano state elaborate e non sia iniziato il round successivo. Di conseguenza, può ordinare le richieste in modo ottimale, probabilmente per cilindro (e in qualche caso per settore), ed elaborarle.

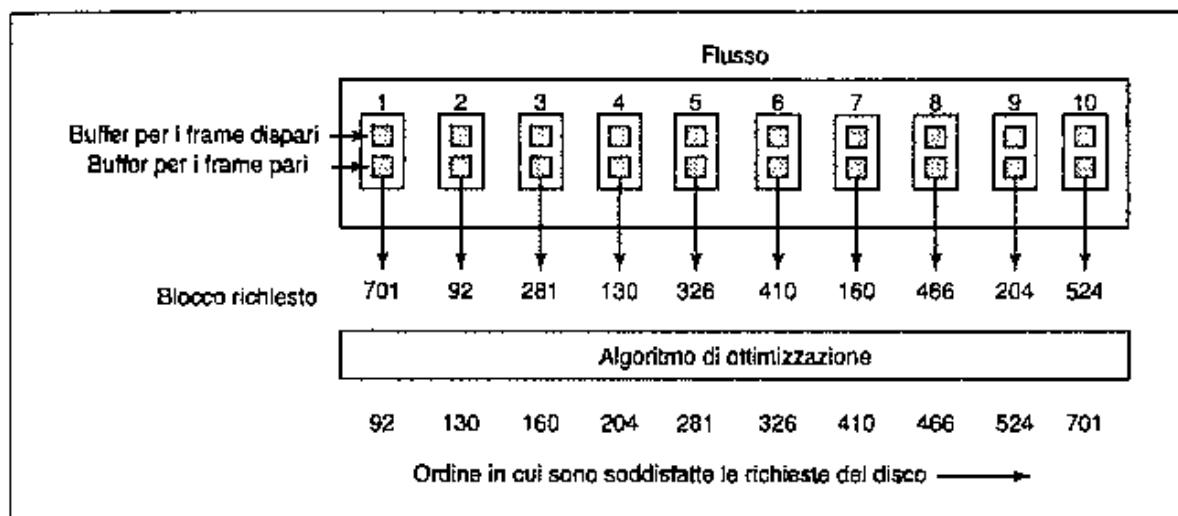


Figura 7.24 In un round ciascun film richiede un frame.

quindi nell'ordine ottimale; nella Figura 7.24 le richieste compaiono ordinate per cilindro.

A prima vista, si potrebbe pensare che ottimizzare un disco in questo modo non serve, perché quando il disco rispetta la scadenza, è irrilevante che lo faccia salvando un millesimo oppure dieci millisecondi. Tuttavia, questa conclusione è falsa: ottimizzando in questo modo i movimenti di posizionamento, si diminuisce il tempo medio per elaborare ciascuna richiesta, cioè il disco può, in media, gestire più flussi per round. In altre parole, ottimizzare le richieste del disco in questo modo incrementa il numero di film che il server può trasmettere simultaneamente. Il tempo risparmiato alla fine del round può anche essere utilizzato per servire eventuali richieste non in tempo reale.

Se un server ha troppi flussi, di tanto in tanto gli è richiesto di prelevare frame da parti distanti del disco, ed è possibile che non rispetti la scadenza; se però le scadenze non rispettate sono abbastanza rare, si possono tollerare, in cambio della gestione di più flussi contemporanei. Si noti che l'importante è il numero di flussi che si stanno gestendo, avere due o più utenti per flusso non ha effetto sulle prestazioni del disco o sulla schedulazione.

Per fare in modo che il flusso dei dati arrivi uniformemente agli utenti, è necessaria la doppia bufferizzazione sul server. Durante il round 1 è usato un insieme di buffer, uno per flusso; quando il round è finito, il processo o i processi di uscita sono sbloccati e trasmettono il frame 1. Nello stesso tempo, arrivano nuove richieste per il frame 2 di ciascun film (potrebbe esserci un thread del disco e un thread di uscita per ciascun film), che sono soddisfatte con un secondo insieme di buffer, poiché i primi sono ancora occupati. Quando inizia il round 3, il primo insieme di buffer è libero e disponibile per prelevare il frame 3.

Abbiamo ipotizzato che ci sia un round per frame, ma questa limitazione non è strettamente necessaria. Potrebbero esserci due round per frame, al fine di ridurre lo spazio necessario per i buffer, ma al costo di duplicare le operazioni del disco. Allo stesso modo, si potrebbero prelevare dal disco due frame per round (ipotizzando che coppie di frame siano memorizzate in modo contiguo sul disco): questo schema riduce della metà il numero delle operazioni del disco, al costo di raddoppiare lo spazio necessario per il buffer. In relazione alla disponibilità relativa delle prestazioni e del costo della memoria rispetto all'I/O del disco, si può calcolare e utilizzare una strategia ottimale.

## 7.8.2 Schedulazione dinamica del disco

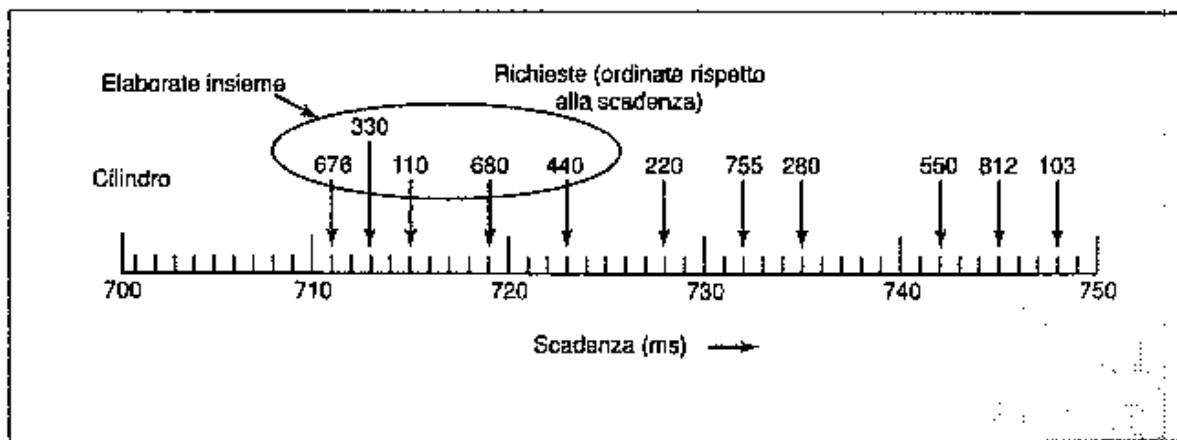
Nell'esempio precedente abbiamo considerato flussi con risoluzione, velocità dei frame e altre proprietà uguali. Abbandoniamo adesso tale ipotesi: film diversi possono avere diverse velocità, cosicché non è possibile avere un round ogni 33,3 ms, e prelevare un frame per ciascun flusso, e le richieste arrivano al disco in modo più o meno casuale.

Ciascuna richiesta di lettura specifica quale blocco debba essere letto, ed inoltre quando si abbia bisogno del blocco, in altre parole la sua scadenza temporale. Per semplicità ipotizzeremo che il tempo di servizio per ciascuna richiesta sia lo stesso (anche se questo non è certamente vero), così possiamo sottrarre il tempo fisso di servizio per ottenere il tempo limite in cui può essere iniziata l'elaborazione della richiesta, rispettando ancora la scadenza temporale. Questo rende il modello più semplice: ciò di cui si occupa lo scheduler del disco è la scadenza temporale per schedulare la richiesta.

Quando il sistema parte, non ci sono richieste al disco, e quando la prima richiesta arriva, è servita immediatamente. Mentre avviene il primo movimento di posizionamento, possono arrivare altre richieste; quando la prima richiesta è soddisfatta, il driver del disco può scegliere quale richiesta elaborare, ed iniziare la sua elaborazione. Finita questa elaborazione, c'è di nuovo un insieme di richieste possibili, quelle non scelte la prima volta, e quelle nuove, giunte nel corso della seconda elaborazione; in generale, quando si completa una richiesta al disco, il driver ha un insieme di richieste pendenti, da cui deve effettuare una scelta. Si presenta un problema: "Quale algoritmo scegliere per servire la richiesta successiva?".

Due fattori giocano un ruolo nella selezione della successiva richiesta: le scadenze temporali e i cilindri; dal punto di vista delle prestazioni, mantenere le richieste ordinate per cilindro e utilizzare l'algoritmo dell'ascensore, minimizza il tempo totale di posizionamento, ma le scadenze temporali per le richieste relative ai cilindri periferici, possono non essere rispettate. Dal punto di vista del tempo reale, ordinare le richieste rispetto alla loro scadenza temporale ed elaborarle secondo tale ordine (prima la richiesta con scadenza più prossima), minimizza la possibilità di non rispettare le scadenze temporali, incrementando però il tempo totale di posizionamento.

È possibile combinare questi fattori usando l'algoritmo scan-EDF (Reddy e Willye, 1992), la cui idea base consiste nel raccogliere in gruppi richieste le cui scadenze temporali siano relativamente vicine, elaborandole in ordine di cilindro. Come esempio, consideriamo la situazione della Figura 7.25 per  $t = 700$ : il driver del disco sa che ci sono undi-



**Figura 7.25** L'algoritmo scan-EDF usa le scadenze temporali e i numeri dei cilindri per la schedulazione.

ci richieste pendenti con diverse scadenze temporali e diversi cilindri. Potrebbe decidere, ad esempio, di trattare le cinque richieste con le scadenze temporali più ravvicinate come un gruppo, ordinarle per numero di cilindro e utilizzare l'algoritmo dell'ascensore per servirle in ordine di cilindro: l'ordine sarebbe 110, 330, 440, 676 e 680. Se ogni richiesta è completata prima della sua scadenza temporale, le richieste possono essere riordinate in sicurezza per minimizzare il tempo totale di posizionamento.

Quando differenti flussi hanno differenti velocità, alla comparsa di un nuovo utente sorge un problema serio: lo si dovrà ammettere? Se l'ammissione del cliente causerà la perdita frequente delle scadenze temporali per gli altri flussi, la risposta è probabilmente negativa. Ci sono due modi per valutare se ammettere o no un nuovo utente; uno consiste nell'ipotizzare che ciascun utente abbia bisogno in media di una certa quantità di risorse, ad esempio larghezza di banda del disco, buffer di memoria, tempo di CPU, eccetera. Se ce n'è a sufficienza per un cliente medio, quello nuovo sarà ammesso.

L'altro algoritmo è più dettagliato, perché tiene in conto il film specifico scelto dal nuovo cliente e considera la velocità di trasmissione dati (precalcolata) per quel film, che differisce per il bianco e nero e il colore, per i cartoni animati e i film, e anche per le storie d'amore e i film di guerra. Le storie d'amore si muovono lentamente con lunghe scene e lente dissolvenze, rendendo possibile una buona compressione; i film di guerra, invece, hanno molti cambiamenti rapidi e azioni veloci, quindi molti frame I e grandi frame P. Se il server ha abbastanza capacità per il film scelto dal nuovo cliente, l'ammissione è garantita, altrimenti, la si rifiuta.

## 7.9 La ricerca sui multimedia

Attualmente, i multimedia sono un argomento di punta, su cui la ricerca concentra un'attenzione rilevante; buona parte di essa riguarda il contenuto, gli strumenti di costruzione e le applicazioni, tutte cose che esulano dai confini di questo libro. Comunque, parte della ricerca riguarda la struttura del sistema operativo: o la scrittura di un nuovo sistema operativo (Brandwein et al., 1994), o l'aggiunta di un supporto multimediale ad un sistema operativo esistente (Mercer, 1994). Un'area collegata è la progettazione dei server multimediali (Bernhardt e Biersack, 1996; Heybey et al., 1996; Louher et al., 1994; Wong e Lee, 1997).

Alcuni articoli sui multimedia non riguardano nuovi sistemi completi, ma algoritmi utili nei sistemi multimediali. Un argomento assai trattato è la schedulazione della CPU in tempo reale per i multimedia (Baker-Harvey, 1999; Bolosky et al., 1997; Dan et al., 1994; Goyal et al., 1996; Jones et al., 1997; Nieh e Lam, 1997; Wu e Shu, 1996). Altro argomento studiato, è la schedulazione del disco per i multimedia (Lee et al., 1997; Rompogiannakis et al., 1998 ; Wang et al., 1999). Sono anche importanti il posizionamento dei file e la gestione dei carico sui video server (Gafsi e Biersack, 1999; Shenoy e Vin, 1999; Shenoy et al., 1999; Venkatasubramanian e Ramanathan, 1997), come lo è il merge dei flussi video per ridurre le esigenze di larghezza di banda (Eager et al., 1999).

Nel testo abbiamo discusso come la popolarità dei film influenzi la loro posizione sul video server; la ricerca su questo argomento si sta sviluppando (Bisdikian e Patel, 1995; Griwadz et al., 1997). Intanto, la ricerca si occupa anche della sicurezza e della tutela della privacy nei multimedia, ad esempio per le videoconferenze (Adams e Sasse, 1999; Honeyman et al., 1998).

## 7.10 Sommario

I multimedia rappresentano un settore emergente nell'utilizzo dei computer; per le grandi dimensioni dei file multimediali e le loro stringenti esigenze di esecuzione in tempo reale, i sistemi operativi progettati per il testo non sono ottimali per i multimedia. I file multimediali sono composti da diverse tracce parallele, di solito una traccia video e almeno una audio, e talvolta anche tracce per i sottotitoli; tutte devono essere sincronizzate durante l'esecuzione.

L'audio è registrato campionando il suono periodicamente, di solito 44100 volte il secondo (per una qualità del suono a livello di CD). Si può applicare la compressione al segnale audio, ottenendo un tasso di compressione uniforme di circa 10 x. La compressione video utilizza sia la compressione all'interno dei frame (JPEG), sia quella tra frame (MPEG); quest'ultima rappresenta i frame P come differenze rispetto al frame precedente; mentre i frame B si basano o sul frame precedente o su quello successivo.

I multimedia hanno bisogno di una schedulazione in tempo reale per rispettare le proprie scadenze temporali. Di solito, si usano due algoritmi: il primo è una schedulazione monotona della velocità, un algoritmo statico con prerilascio, che assegna priorità fisse ai processi basate sulla loro durata; il secondo è basato sul principio di servire per prime le scadenze temporali più stringenti: è un algoritmo dinamico, che sceglie sempre il processo con la scadenza temporale più vicina. EDF è più complicato ma può arrivare ad un utilizzo del 100%, cosa che RMS non può ottenere.

I file system multimediali di solito usano un modello "push" piuttosto che "pull": una volta che un flusso è partito, i bit lasciano il disco senza ulteriori richieste dell'utente. Questo approccio è assai diverso da quello dei sistemi operativi convenzionali, ma è necessario per soddisfare le esigenze del tempo reale.

I file possono essere memorizzati in modo contiguo o no; nel secondo caso, le unità possono essere di lunghezza variabile (un blocco è un frame), o di lunghezza fissa (un blocco è costituito da molti frame). Questi approcci hanno differenti trade off.

Il posizionamento dei file sul disco influenza le prestazioni; quando ci sono file differenti, talvolta si usa l'algoritmo a canne d'organo. È comune effettuare suddivisioni dei file fra più dischi, sia a fette grandi, sia a fette piccole. Si impiegano anche strategie di cache per i blocchi e i file, per migliorare le prestazioni.

## PROBLEMI

1. Qual è il tasso di trasferimento dei bit per un flusso XGA a pieni colori non compresso, eseguito a 25 frame/s? Può un flusso a questa velocità provenire da un disco SCSI Ultra Wide?
2. Si può inviare su una fast Ethernet una trasmissione televisiva NTSC non compressa in bianco e nero? Se sì, quanti canali insieme?
3. HDTV ha due volte la risoluzione orizzontale di una TV regolare (1280 pixel invece di 640 pixel). Utilizzando l'informazione contenuta nel testo, quanta larghezza di banda in più è richiesta rispetto ad una TV standard?
4. Nella Figura 7.3 ci sono file separati per avanti veloce e indietro veloce. Se un video server vuole supportare anche l'esecuzione rallentata, è richiesto un altro file ad hoc per l'esecuzione rallentata in avanti? E per l'esecuzione rallentata all'indietro?

5. Un Compact Disk memorizza 74 minuti di musica o 650 MB di dati. Fai una stima del fattore di compressione utilizzato per la musica.
6. Un segnale sonoro è campionato tramite un numero con segno a 16-bit (1 bit di segno, 15 bit per l'ampiezza). A quanto ammonta in percentuale il rumore di quantizzazione massimo? È un problema maggiore per i concerti di flauto, o di rock and roll, o è lo stesso per entrambi? Si motivi la risposta.
7. Uno studio di registrazione è in grado di produrre un master di registrazione digitale con campionamento a 20-bit.  
La distribuzione finale agli ascoltatori utilizzerà 16 bit. Si suggerisca un modo per ridurre l'effetto del rumore di quantizzazione, e si discutano i vantaggi e gli svantaggi dello schema proposto.
8. NTSC e PAL utilizzano entrambi un canale di trasmissione a 6 MHz, tuttavia NTSC ha 30 frame/s, mentre PAL ha solo 25 frame/s.  
Com'è possibile? Questo significa che se entrambi i sistemi usassero lo stesso schema di codifica del colore, NTSC avrebbe intrinsecamente una migliore qualità rispetto a PAL? Si motivi la risposta.
9. La trasformazione DCT utilizza un blocco 8 x 8, tuttavia l'algoritmo usato per la compensazione del movimento utilizza 16 x 16. Questa differenza causa problemi, e, se è così, come sono risolti in MPEG?
10. Nella Figura 7.10 abbiamo visto come lavori MPEG con sfondo fisso, e attore in movimento. Si supponga che un video MPEG sia fatto da una scena in cui la telecamera è montata su un piedistallo, ed effettua una panoramica lenta da sinistra a destra, ad una velocità tale che 2 frame consecutivi non siano uguali. Tutti i frame devono essere frame I? Perché sì o perché no?
11. Si supponga che ciascuno dei tre processi della Figura 7.11 sia accompagnato da un processo che supporta un flusso audio eseguito contemporaneamente al processo video, cosicché i buffer audio possano essere aggiornati tra i frame video.  
Tutti e tre i processi audio sono identici. Quanto tempo di CPU è disponibile per ciascun burst di un processo audio?
12. Due processi in tempo reale sono in esecuzione su un computer. Il primo viene eseguito ogni 25 ms per 10 ms. Il secondo ogni 40 ms per 15 ms. RMS funzionerà sempre per essi?
13. La CPU di un video server ha un'utilizzazione del 65%. Quanti film può visualizzare utilizzando la schedulazione RMS?
14. Nella Figura 7.13 EDF mantiene la CPU occupata al 100% del tempo, fino a  $t = 150$ . Non può mantenere la CPU occupata indefinitivamente, perché dispone solo di 975 ms di lavoro ogni secondo. Si estenda la figura oltre i 150 ms e si determini quando la CPU rimarrà inutilizzata per la prima volta con EDF.
15. Un DVD può contenere abbastanza dati per un film a lunghezza piena, e il tasso di trasferimento è adeguato per visualizzare un programma di qualità televisiva.  
Perché non utilizzare una "farm" di molti drive DVD come sorgente di dati per un video server?
16. Gli operatori di un sistema video quasi su richiesta, hanno scoperto che le persone in una determinata città non vogliono aspettare più di sei minuti l'inizio di un film. Di quanti flussi paralleli hanno bisogno per un film di tre ore?

17. Si consideri un sistema che usi lo schema di Abram-Profeta e Shin, in cui l'operatore del video server consenta agli utenti di cercare avanti e indietro per un minuto interamente in locale. Se il flusso video è un MPEG-2 a 4 bps, quanto spazio buffer deve possedere ciascun utente localmente?
18. Un sistema di video su richiesta per HDTV utilizza il modello a blocchi piccoli della Figura 7.18 (a), con blocco del disco di 1 KB. Se la risoluzione video è 1280 x 720 e il flusso dei dati è a 12 Mbps, quanto spazio su disco viene sprecato per frammentazione interna in un film di due ore che utilizzi NTSC?
19. Si consideri lo schema di allocazione di memoria della Figura 7.18 (a) per NTSC e PAL. Per una data dimensione del blocco del disco e del film, uno di essi subirà più frammentazione interna dell'altro? Se sì, quale è meglio e perché?
20. Si considerino le due possibilità mostrate in Figura 7.18. Il passaggio a HDTV favorisce l'uno o l'altro di questi due sistemi? Si discuta il problema.
21. Lo schema di video quasi su richiesta di Chen e Thapar funziona meglio quando ciascun insieme di frame è della stessa dimensione. Si supponga che si stia mostrando un film in 24 flussi simultanei, e che un frame ogni dieci sia un frame I. Ancora si supponga che i frame I siano dieci volte più grandi dei frame P, ed i frame B abbiano le stesse dimensioni dei frame P. Qual è la probabilità che un buffer grande come 4 frame I e 20 frame P non sia grande abbastanza? Si ritiene che la dimensione di tale buffer sia accettabile? Per rendere il problema trattabile, si supponga che i vari tipi di frame siano casualmente e indipendentemente distribuiti sui flussi.
22. Il risultato finale della Figura 7.16 è che il punto di esecuzione non è più nel mezzo del buffer. Si trovi uno schema che abbia il punto di esecuzione almeno 5 minuti prima e 5 minuti dopo di esso. Si formulino ragionevoli ipotesi, a condizione di enunciarle esplicitamente.
23. Il progetto della Figura 7.17 richiede che tutte le tracce del parlato siano lette sullo stesso frame. Si supponga che i progettisti di un video server debbano supportare un gran numero di lingue, ma non vogliano dedicare tanta RAM ai buffer in cui tenere i frame. Quali altre possibilità esistono? Con quali svantaggi e vantaggi?
24. Un piccolo video server ha otto film. Che cosa predirà la legge di Zipf per le probabilità del più popolare, del secondo, e così via sino al film meno popolare?
25. Un disco da 14 GB con 1000 cilindri è utilizzato per mantenere 1000 video clip MPEG-2 da 30 secondi, eseguiti a 4 Mbps. Essi sono memorizzati secondo l'algoritmo a canne d'organo. In base alla legge di Zipf, per quale frazione di tempo il braccio del disco rimarrà nei dieci cilindri centrali?
26. Si supponga che la richiesta relativa per i film A, B, C e D sia descritta dalla legge di Zipf. Quale sarà l'utilizzo relativo atteso dei quattro dischi della Figura 7.22 per i quattro metodi di suddivisione mostrati?
27. Due utenti di un video su richiesta hanno iniziato a guardare lo stesso film PAL a sei secondi di distanza l'uno dall'altro. Se il sistema accelera un flusso e decelera l'altro per effettuare il merge, quale percentuale di accelerazione/decelerazione è necessaria per ottenere il merge in tre minuti?
28. Un video server MPEG-2 utilizza lo schema round della Figura 7.24 per i video NTSC. Tutti i video provengono da un singolo disco SCSI UltraWide a 10800 rpm, con un tempo medio di ricerca di 3 ms. Quanti flussi possono essere supportati?

29. Si ripeta il precedente problema, ipotizzando che scan-EDF riduca il tempo medio di ricerca del 20%. Quanti flussi possono essere ora supportati?
30. Si ripeta ancora il problema precedente, ipotizzando che ciascun frame sia suddiviso fra quattro dischi, con scan-EDF che fa guadagnare il 20% a ciascun disco. Quanti flussi possono essere ora supportati?
31. Il testo descrive l'utilizzo di un gruppo di cinque richieste di dati, per schedulare la situazione descritta nella Figura 7.25 (a). Se tutte le richieste prendono un'uguale quantità di tempo, qual è il tempo massimo per richiesta disponibile in questo esempio?
32. Molte delle immagini bitmap per generare gli sfondi dei computer, utilizzano pochi colori, e si comprimono facilmente. Un semplice schema di compressione è il seguente: si scelga un valore dei dati che non appare nel file di input, e lo si utilizzi come flag. Si legga il file, byte per byte, cercando valori di byte ripetuti. Si copino valori singoli e byte ripetuti fino a tre volte direttamente nel file di uscita, ma quando si trova una stringa ripetuta di 4 o più byte, si scriva nel file di uscita una stringa di 3 byte, costituita da 1 byte di flag, 1 byte che indica un contatore da 4 a 255, e il valore effettivo trovato nel file di input. Si scriva un programma di compressione che usi questo algoritmo, e un programma di decompressione in grado di ricostituire il file originale. Punteggio extra: come si possono trattare file che contengono il byte di flag nei loro dati?
33. L'animazione computerizzata è attuata visualizzando una sequenza di immagini leggermente differenti. Si scriva un programma per calcolare la differenza byte per byte fra due immagini bitmap non compresse delle stesse dimensioni. Il file di uscita sarà delle stesse dimensioni di quello di entrata, naturalmente; si usi questo file differenza come ingresso del programma di compressione del problema precedente, e si confronti l'efficacia di questo approccio con la compressione di immagini singole.

# SISTEMI CON PROCESSORI MULTIPLI

Sin dagli inizi, l'industria dei computer è stata guidata dalla continua ricerca di una sempre maggior potenza di calcolo. L'ENIAC poteva eseguire trecento operazioni al secondo, ad una velocità senza dubbio mille volte superiore a quella di ogni calcolatore precedente, tuttavia, non si era soddisfatti; ora abbiamo macchine un milione di volte più veloci dell'ENIAC, e ancora c'è richiesta di maggior potenza. Gli astronomi stanno cercando di dare un significato all'universo, i biologi cercano di comprendere il genoma umano, gli ingegneri aeronautici sono interessati a costruire aerei più sicuri ed efficienti, e tutti vogliono più cicli di CPU. In ogni modo, per quanto grande sia la potenza di cui si può disporre, non ce n'è mai abbastanza.

In passato, la soluzione consisteva sempre nell'aumentare la velocità del clock. Sfortunatamente, quanto alla velocità del clock, stiamo raggiungendo alcuni limiti strutturali. In accordo con la teoria della relatività ristretta di Einstein, nessun segnale elettrico si può propagare più velocemente della velocità della luce, che è di circa 30 cm/ns nel vuoto, e di circa 20 cm/ns nei cavetti di rame o nelle fibre ottiche. Questo significa che in un computer con un clock di 10 GHz, i segnali non possono attraversare più di due centimetri in totale; per un computer a 100 GHz, la lunghezza totale del cammino è al massimo di 2 mm. Ad 1 THz (1000 GHz), un computer dovrà essere più piccolo di 200 micron, per consentire al segnale di andare e tornare da un capo all'altro entro un singolo ciclo di clock.

Rendere i computer così piccoli è possibile, ma ci si trova però di fronte ad un altro problema fondamentale, la dispersione del calore. Più veloce va il computer, più calore genera, e più piccolo è il computer, più difficile è liberarsi di questo calore. Già nei Pentium di fascia alta, il sistema di raffreddamento della CPU è più grande della CPU stessa. Tutto sommato, passare da 1 MHz a 1 GHz richiede una migliore ingegnerizzazione

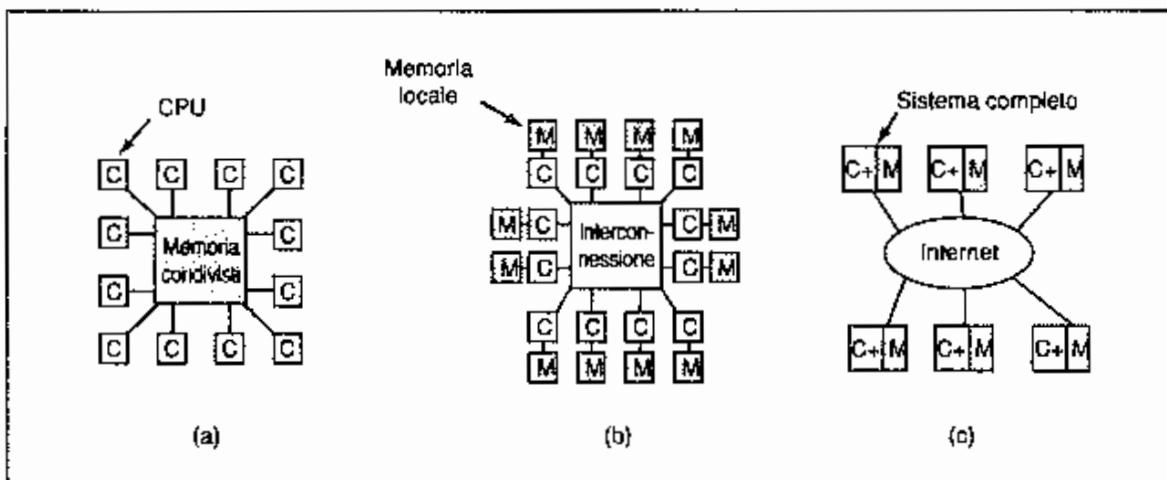
dei processi di fabbricazione dei chip, ma passare da 1 Ghz a 1 THz richiede un approccio radicalmente diverso.

Un modo di ottenere una maggior velocità consiste nell'impiego di computer massicciamente paralleli. Queste macchine si compongono di molte CPU, ciascuna delle quali gira a velocità "normale" (qualunque cosa ciò significhi in un determinato momento), ma collettivamente possiedono una potenza di calcolo molto superiore a quella di una singola CPU. Attualmente, si trovano in commercio sistemi con mille CPU; ed è probabile che, nel prossimo decennio, si costruiranno sistemi con un milione di CPU. Esistono altri potenziali approcci per ottenere una maggior velocità, come i computer biologici, ma in questo capitolo tratteremo i sistemi con molte CPU convenzionali.

Computer altamente paralleli sono spesso usati per il calcolo numerico pesante: problemi come le previsioni meteorologiche, la modellazione del flusso dell'aria attorno all'ala di un aereo, la simulazione dell'economia mondiale, o la comprensione delle interazioni tra farmaco e recettore nel cervello, sono tutti computazionalmente gravosi. La loro soluzione richiede lunghi tempi di calcolo su molte CPU insieme: i sistemi a processori multipli discussi in questo capitolo sono ampiamente usati per questi e simili problemi in campo scientifico e ingegneristico, per non parlare di altri settori.

Un altro sviluppo rilevante consiste nella crescita incredibilmente rapida di Internet. In origine, Internet è stata progettata come prototipo di un sistema di controllo militare resistente a guasti; poi si è diffusa nella comunità dei ricercatori informatici, e di recente ha acquisito molti nuovi usi. Uno di questi consiste nel collegare migliaia di computer in tutto il mondo per lavorare insieme su problemi scientifici di ampie dimensioni. In un certo senso, un sistema costituito da mille computer dislocati in tutto il mondo, non è diverso da uno costituito da mille macchine in un'unica stanza, benché i ritardi di trasmissione, e altre caratteristiche tecniche non siano uguali. In questo capitolo, prenderemo in esame anche questi sistemi.

Sistemare un milione di computer non collegati in una stanza è facile, a patto di avere abbastanza denaro, e una stanza grande a sufficienza. Sparagliare ovunque un milione di computer non collegati, è ancora più facile, perché si evita il secondo problema. I problemi si presentano quando li si vuole mettere in comunicazione tra di loro, per lavorare insieme su di un determinato problema. Di conseguenza, è stato fatto un grande sforzo per elaborare una tecnologia di interconnessione, e molte di tali tecnologie hanno prodotto sistemi di qualità diversa, e diverse organizzazioni software.



**Figura 8.1** (a) Multiprocessore a memoria condivisa. (b) Multicomputer a scambio di messaggi. (c) Un sistema distribuito geografico.

Tutte le comunicazioni fra componenti elettroniche (oppure ottiche) si riducono, in ultima analisi, a mandare messaggi (stringhe di bit ben definite) fra le componenti stesse. Le differenze sono nella scala dei tempi, delle distanze, e nell'organizzazione logica: ad un estremo, ci sono i multiprocessori a memoria condivisa, sistemi dove comunicano da 2 a circa 1000 CPU, mediante una memoria condivisa; in questo modello, ogni CPU ha uguali possibilità di accesso all'intera memoria fisica, e può leggere e scrivere parole singole, tramite le istruzioni LOAD e STORE; tale accesso richiede di solito 10-50 ns. Mentre questo modello, illustrato nella Figura 8.1 (a) sembra semplice, di fatto implementarlo non lo è per nulla, e di solito comporta un impiego nascosto di scambio di messaggi, come spiegheremo brevemente.

In seguito, si esaminerà il sistema illustrato dalla Figura 8.1 (b), in cui un insieme di coppie memoria-CPU sono connesse da un qualche sistema di interconnessione ad alta velocità. Questo tipo di sistema è chiamato multicomputer a scambio di messaggi: ciascuna memoria è locale ad una sola CPU, e vi si può accedere solo tramite la sua CPU; le macchine comunicano mandando messaggi di più parole attraverso il sistema di interconnessione. Con una buona interconnessione, un messaggio breve può essere spedito in 10-50  $\mu$ s, ma siamo ancora lontani dal tempo di accesso alla memoria della Figura 8.1 (a); in questa architettura non esiste memoria globale condivisa. I multicomputer (cioè i sistemi a scambio di messaggi) sono molto più facili da costruire dei multiprocessori a memoria condivisa, ma più difficili da programmare. Quindi, ciascun modello ha i suoi estimatori.

Il terzo modello, illustrato nella Figura 8.1(c), connette sistemi di computer completi su una rete geografica, come Internet, per formare un **sistema distribuito**. Ciascuno di questi ha la sua memoria, ovviamente, e i sistemi comunicano per scambio di messaggi. La sola vera differenza tra la Figura 8.1 (c) e la Figura 8.1 (b) è che nel primo caso sono utilizzati computer completi, e i tempi dei messaggi sono spesso di 10-50 ms. Questi notevoli ritardi forzano i sistemi ad accoppiamento lasso (*loosely coupled*) ad essere utilizzati in modi diversi rispetto a quelli ad accoppiamento stretto (*tightly coupled*) della Figura 8.1 (b). I tre tipi di sistema differiscono nei loro ritardi per ben tre ordini di grandezza. La differenza, cioè, fra un giorno e tre anni.

Questo capitolo contiene tre sezioni principali, corrispondenti ai tre modelli della Figura 8.1. Ciascuna inizia con una breve descrizione dell'hardware pertinente al sistema, per passare poi al software, in particolare alle questioni inerenti al sistema operativo. Come vedremo, in ciascun caso si presentano problemi diversi.

## 8.1 Multiprocessori

Un multiprocessore a memoria condivisa (d'ora in poi, multiprocessore, abbreviato SMP, cioè Shared-memory MultiProcessor) è un sistema di computer in cui due o più CPU condividono il pieno accesso ad una RAM comune. Un programma in esecuzione su una delle CPU utilizza un normale spazio degli indirizzi virtuale (di solito paginato); la sola proprietà peculiare di questi sistemi è che la CPU può scrivere qualche valore in una parola di memoria, e quando la rilegge può ottenere un valore differente (perché un'altra CPU lo ha cambiato). Se organizzata correttamente, questa proprietà fornisce le basi della comunicazione fra processi: una CPU scrive dati in memoria, un'altra li legge.

Per la maggior parte, i sistemi operativi multiprocessore sono sistemi operativi normali: gestiscono le chiamate di sistema, la memoria e i dispositivi di I/O, e forniscono un file system. Tuttavia, ci sono aree in cui sono dotati di caratteristiche proprie, ad esempio la sincronizzazione dei processi, la gestione delle risorse e la schedulazione. In seguito,

daremo un rapido sguardo all'hardware dei multiprocessori, per dedicarci poi alle caratteristiche di questi sistemi operativi.

### 8.1.1 Hardware multiprocessore

Benché tutti i multiprocessori siano dotati della proprietà che ogni CPU possa indirizzare la memoria, alcuni processori hanno un'ulteriore proprietà: ogni parola di memoria può essere letta tanto velocemente quanto ogni altra; tali macchine sono chiamate multiprocessori UMA (Uniform Memory Access, accesso uniforme alla memoria). Al contrario, i multiprocessori NUMA (Non Uniform Memory Access, accesso non uniforme alla memoria) non possiedono questa proprietà. Chiariremo in seguito perché esista tale differenza, ma prima esamineremo i multiprocessori UMA, per poi passare ai NUMA.

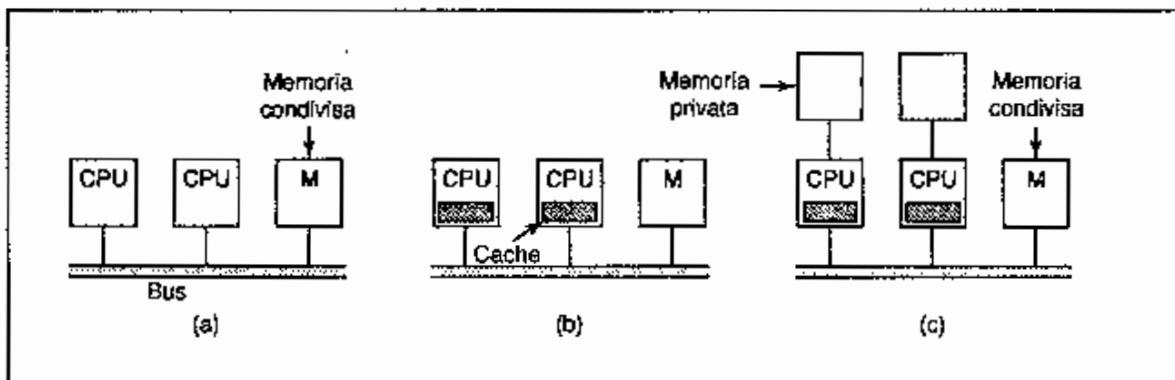
#### Architetture SMP UMA basate sul bus

I multiprocessori più semplici sono basati su un singolo bus, come si vede nella Figura 8.2 (a): due o più CPU, e uno o più moduli di memoria utilizzano per la comunicazione tutti lo stesso bus. Quando una CPU vuol leggere una parola di memoria, prima controlla se il bus è occupato; se è libero, la CPU pone l'indirizzo della parola richiesta sul bus, attiva alcuni segnali di controllo e aspetta finché la memoria pone la parola desiderata sul bus.

Se il bus è occupato quando una CPU vuole leggere o scrivere in memoria, la CPU aspetta fino a che il bus è libero, e questo è il problema inerente tale architettura: con due o tre CPU, la contesa per il bus sarà gestibile, ma con 32 o 64 diventerà insostenibile. Il sistema sarà totalmente limitato dalla larghezza di banda del bus, e la maggior parte delle CPU rimarrà inattiva per gran parte del tempo.

La soluzione di questo problema consiste nell'aggiungere una cache a ciascuna CPU, come illustra la Figura 8.2 (b); essa può essere sul chip della CPU, o vicino al suo chip, o sulla scheda del processore oppure in una combinazione di tutti e tre. Poiché molte letture possono ora essere soddisfatte dalle cache locali, ci sarà meno traffico sul bus, e il sistema potrà supportare un maggior numero CPU. In generale, il meccanismo di cache non è fatto sulla base di una parola singola, ma sulla base di blocchi di 32 o 64 byte: quando si fa riferimento ad una parola, tutto il suo blocco è prelevato nella cache della CPU che lo ha richiesto.

Ciascun blocco della cache è marcato per essere o in sola lettura (in tal caso può esse-



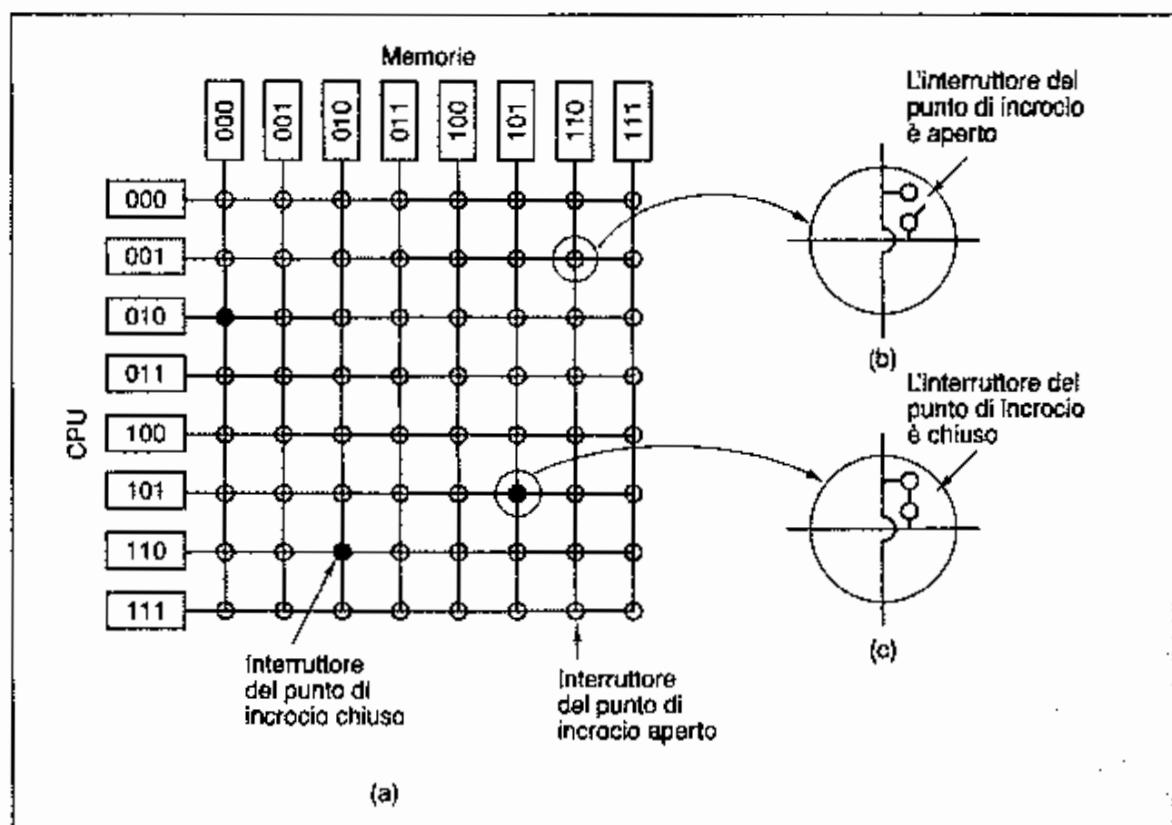
**Figura 8.2** Tre multiprocessori basati sul bus. (a) Senza meccanismo di cache. (b) Con la cache. (c) Con la cache e memorie private.

re presente in diverse cache nello stesso momento), o in lettura-scrittura (può non essere presente in altre cache). Se una CPU tenta di scrivere una parola contenuta in una o più cache remote, l'hardware del bus intercetta la scrittura ed emette un segnale sul bus informando tutte le altre cache della scrittura. Se le altre cache hanno una copia "pulita", (cioè, una copia esatta di ciò che si trova in memoria), possono scaricarla, e permettono alla CPU scrittrice di prelevare il blocco della cache dalla memoria prima di modificarlo. Se qualche altra cache ha una copia "sporca" (cioè modificata), deve riscriverla in memoria, prima che la scrittura inizi, oppure trasferirla direttamente alla CPU scrittrice sul bus. Esistono molti protocolli di trasferimento della cache.

Un'altra possibilità è l'architettura della Figura 8.2 (c), in cui ciascuna CPU ha non solo una cache, ma anche una memoria locale e privata, cui accede tramite un bus dedicato (privato). Per utilizzare al meglio questa configurazione, il compilatore dovrebbe posizionare tutto il testo del programma, le stringhe, le costanti e gli altri dati in sola lettura, gli stack e le variabili locali nelle memorie private; la memoria condivisa allora sarebbe usata solo per le variabili condivise scrivibili. Nella maggior parte dei casi, un tale accurato posizionamento ridurrà di molto il traffico sul bus, richiedendo però un'attiva collaborazione da parte del compilatore.

### Multiprocessori UMA che utilizzano gli interruttori Crossbar

Anche con il miglior meccanismo di cache, l'uso di un bus singolo limita la dimensione dei multiprocessori UMA a circa 16 o 32 CPU; per andare oltre, occorre un diverso tipo di interconnessione di rete. Il circuito più semplice per connettere  $n$  CPU a  $k$  memorie, è



**Figura 8.3** (a) Un interruttore crossbar 8 x 8. (b) Un punto di incrocio aperto. (c) Un punto di incrocio chiuso.

L'interruttore crossbar, illustrato in Figura 8.3. Gli interruttori crossbar sono stati impiegati per decenni negli scambi di interconnessione telefonica, per connettere un gruppo di linee in ingresso ad un insieme di linee in uscita in modo arbitrario.

A ciascuna intersezione di una linea orizzontale (entrante) con una verticale (usciente), c'è un punto di incrocio; esso è un piccolo interruttore, che può essere aperto o chiuso elettricamente, a seconda se le linee orizzontali e verticali debbano essere connesse, oppure no. Nella Figura 8.3 (a) si vedono tre punti di incrocio chiusi simultaneamente, che permettono le connessioni fra le coppie CPU-memoria (010, 000), (101, 101) e (110, 010) nello stesso tempo. Sono anche possibili molte altre combinazioni; infatti, il numero di combinazioni è uguale al numero di diverse posizioni in cui otto torri possono essere sistemate sulla scacchiera, senza attaccarsi reciprocamente.

Una delle migliori proprietà dell'interruttore crossbar consiste nell'essere una rete non bloccante il che significa che non viene mai negata a nessuna CPU la connessione di cui ha bisogno, se alcuni punti di incrocio o la linea sono già occupati (supponendo che il modulo di memoria stesso sia disponibile). Inoltre, non occorre una pianificazione preventiva: anche se sette connessioni arbitrarie sono già attive, è sempre possibile connettere la rimanente CPU alla memoria che rimane.

Una delle peggiori proprietà dell'interruttore crossbar, è che il numero dei punti di incrocio cresce come  $n^2$ : con mille CPU e mille moduli di memoria, si ha bisogno di un milione di punti di incrocio. Un interruttore crossbar così grande non può esistere, mentre per sistemi di medie dimensioni l'architettura crossbar si può utilizzare.

## Multiprocessori UMA che utilizzano reti di interconnessione a più stadi

Un'architettura multiprocessore completamente diversa è basata sulla semplice connessione  $2 \times 2$ , come in Figura 8.4 (a); questa connessione ha due ingressi e due uscite, ed i messaggi che arrivano ad una qualunque delle linee di ingresso possono essere smistati a ciascuna delle linee di uscita. Ai nostri fini, i messaggi conterranno fino a quattro parti, come mostrato in Figura 8.4 (b): il campo Modulo indica quale memoria usare, l'Indirizzo specifica un indirizzo entro il modulo, il Codice Operativo indica l'operazione, come READ, o WRITE; ed infine, il campo opzionale Valore può contenere un operando cioè una parola a 32 bit che dev'essere scritta con un WRITE. La connessione controlla il campo modulo e lo utilizza per determinare se il messaggio dev'essere mandato a X o Y.

Le nostre connessioni  $2 \times 2$  possono essere sistematiche in molti modi, per costruire più ampie reti di connessione a più stadi (Adams et al., 1987; Bhuyan et al., 1989; Kumar e Reddy, 1987). Una possibilità è la semplice ed economica classe delle reti omega, illustrata nella Figura 8.5. Qui abbiamo otto CPU connesse a otto memorie, attraverso dodici connessioni; più in generale, per  $n$  CPU e  $n$  memorie, sono necessari  $\log_2 n$  stadi con  $n/2$

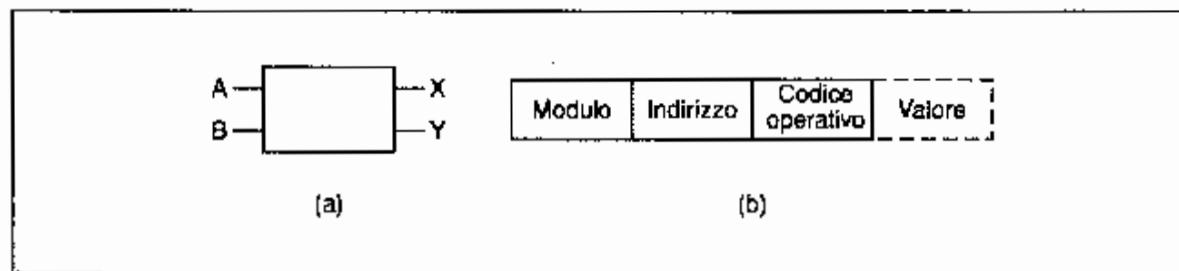


Figura 8.4 (a) Una connessione  $2 \times 2$ . (b) Il formato di un messaggio.

connessioni per stadio, per un totale di  $(n/2) \log_2 n$  connessioni, che è molto meglio di  $n^2$  punti di incrocio, specialmente per grandi valori di  $n$ .

Lo schema di collegamento della rete omega è spesso chiamato **mescolamento perfetto** (*perfect shuffle*), poiché la miscela dei segnali in ciascuno stadio ricorda un mazzo di carte tagliato e quindi mescolato carta per carta. Per vedere come funziona una rete omega, si supponga che la CPU 011 voglia leggere una parola dal modulo di memoria 110. La CPU manda un messaggio READ alla connessione 1D, contenente 110 nel campo Modulo. Lo switch prende il primo bit di 110 (cioè quello più a sinistra) e lo usa per l'instradamento. Uno 0 conduce all'uscita più alta, e un 1 conduce a quella più bassa. Poiché questo bit è un 1, il messaggio è instradato attraverso l'uscita più bassa, verso 2D.

Tutte le connessioni del secondo stadio, tra cui 2D, utilizzano il secondo bit per l'instradamento; esso è ancora un 1, cosicché il messaggio è ora spedito attraverso l'uscita più bassa a 2D. Qui viene testato il terzo bit, e si trova che è uguale a 0, di conseguenza, il messaggio passa attraverso l'uscita superiore e arriva alla memoria 110 come desiderato. Il cammino seguito da questo messaggio è marcato nella Figura 8.5 dalla lettera *a*.

Man mano che il messaggio si muove attraverso le reti di connessione, i bit della parte sinistra del numero del modulo non sono più necessari. È possibile in ogni caso utilizzarli, registrando nella rete di interconnessione il numero di linea entrante, in modo che la risposta possa trovare la propria via al ritorno. Per il cammino *a*, le linee entranti sono 0 (input superiore per 1D), 1 (input inferiore per 2D) e 1 (input inferiore per 3D) rispettivamente. La risposta è instradata all'indietro utilizzando 011, leggendolo questa volta semplicemente da destra a sinistra.

Nello stesso tempo in cui tutto questo sta avvenendo, la CPU 001 vuole scrivere una parola nel modulo di memoria 001. Qui avviene un processo analogo, con il messaggio instradato rispettivamente attraverso le uscite superiore, superiore e inferiore, marcate dalla lettera *b*; quando arriva, il suo campo *Modulo* contiene 001, che rappresenta il cammino che ha seguito. Poiché queste due richieste non utilizzano le stesse connessioni, linee o moduli di memorie, possono procedere in parallelo.

Consideriamo ora che cosa accadrebbe se la CPU 000 volesse contemporaneamente accedere al modulo di memoria 000: la sua richiesta cadrebbe in conflitto con quella della CPU 001 a livello della connessione 3°, ed una di esse dovrebbe aspettare. A differenza dell'interruttore crossbar, la rete omega è una rete bloccante, cioè non tutti gli insiemi

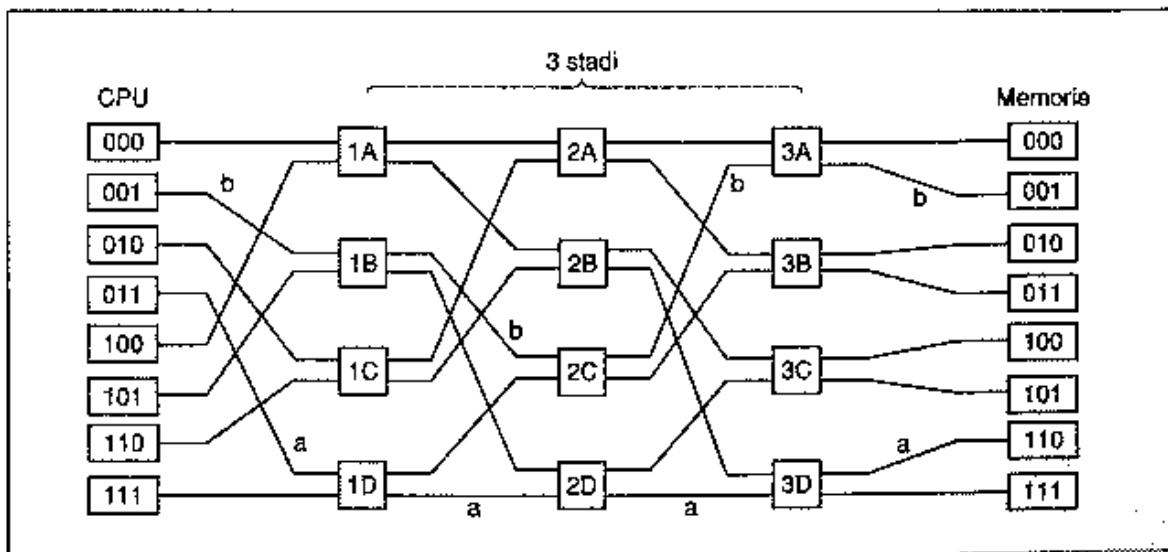


Figura 8.5 Una rete di connessione omega.

di richieste possono essere elaborati simultaneamente, perché possono verificarsi conflitti per l'uso di una linea o di una connessione, sia per richieste verso la memoria, sia per risposte dalla memoria.

È chiaramente desiderabile diffondere i riferimenti alla memoria in modo uniforme per tutti i moduli; una tecnica comune consiste nell'utilizzare i bit di ordine basso come numeri del modulo. Si consideri per esempio uno spazio degli indirizzi orientato ai byte per un computer che il più delle volte accede a parole a 32 bit: i due bit di ordine basso saranno di solito 00, ma i successivi tre bit saranno uniformemente distribuiti, ed utilizzando questi tre bit come numero del modulo, parole indirizzate consecutive saranno in moduli consecutivi. I sistemi di memoria in cui parole consecutive sono in moduli differenti, sono detti interallacciati; le memorie interallacciate massimizzano il parallelismo, perché la maggior parte dei riferimenti a memoria riguardano indirizzi consecutivi. È anche possibile progettare reti di connessione che siano non bloccanti e che offrano più cammini da ciascuna CPU a ciascun modulo di memoria, per distribuire meglio il traffico.

## Multiprocessori NUMA

I multiprocessori UMA a bus singolo sono generalmente limitati a non più di poche dozzine di CPU; i multiprocessori crossbar o basati su connessioni, richiedono molto hardware (costoso), e non sono molto più grandi. Per avere più di cento CPU, si deve rinunciare a qualcosa; di solito, si rinuncia all'idea che tutti i moduli di memoria abbiano lo stesso tempo di accesso. Questa concessione porta all'idea dei multiprocessori NUMA, come accennato prima: a differenza dei loro cugini UMA, forniscono un singolo spazio degli indirizzi a tutte le CPU; ma diversamente dalle macchine UMA, l'accesso ai moduli della memoria locale è più veloce rispetto all'accesso ai moduli remoti. Quindi, tutti i programmi UMA saranno eseguibili senza cambiamento sulle macchine NUMA, ma le prestazioni saranno peggiori rispetto a quelle di una macchina UMA, alla stessa velocità di clock.

Le macchine NUMA hanno tre caratteristiche chiave, che le distinguono dagli altri multiprocessori:

1. C'è un singolo spazio degli indirizzi visibile a tutte le CPU.
2. L'accesso alla memoria remota avviene tramite le istruzioni LOAD e STORE.
3. L'accesso alla memoria remota è più lento rispetto all'accesso alla memoria locale.

Quando il tempo di accesso alla memoria remota non è nascosto (perché non c'è cache), il sistema è chiamato NC-NUMA. Quando sono presenti cache coerenti, si parla di CC-NUMA (Cache-Coherent NUMA).

Il più seguito approccio per costruire grandi sistemi multiprocessore CC-NUMA è attualmente il multiprocessore basato su directory. L'idea è di mantenere un database che dica dove sia ciascuna linea di cache, e quale sia il suo stato; quando si ha un riferimento ad una linea di cache, s'interroga il database, perché trovi dov'è e se sia pulita o sporca (modificata). Poiché questo database dev'essere interrogato per ogni istruzione che faccia riferimento alla memoria, occorre mantenerlo in un hardware dedicato estremamente veloce, che possa rispondere in una frazione di ciclo di bus.

Per farsi un'idea un po' più concreta di un multiprocessore basato su directory, si consideri un semplice (ipotetico) esempio, un sistema a 256 nodi, dove ciascun nodo è costituito da una CPU, e 16 MB di RAM connessi alla CPU attraverso un bus locale. La memoria totale è di  $2^{32}$  byte, divisa in  $2^{26}$  linee di cache di 64 byte ciascuna, ed è staticamente

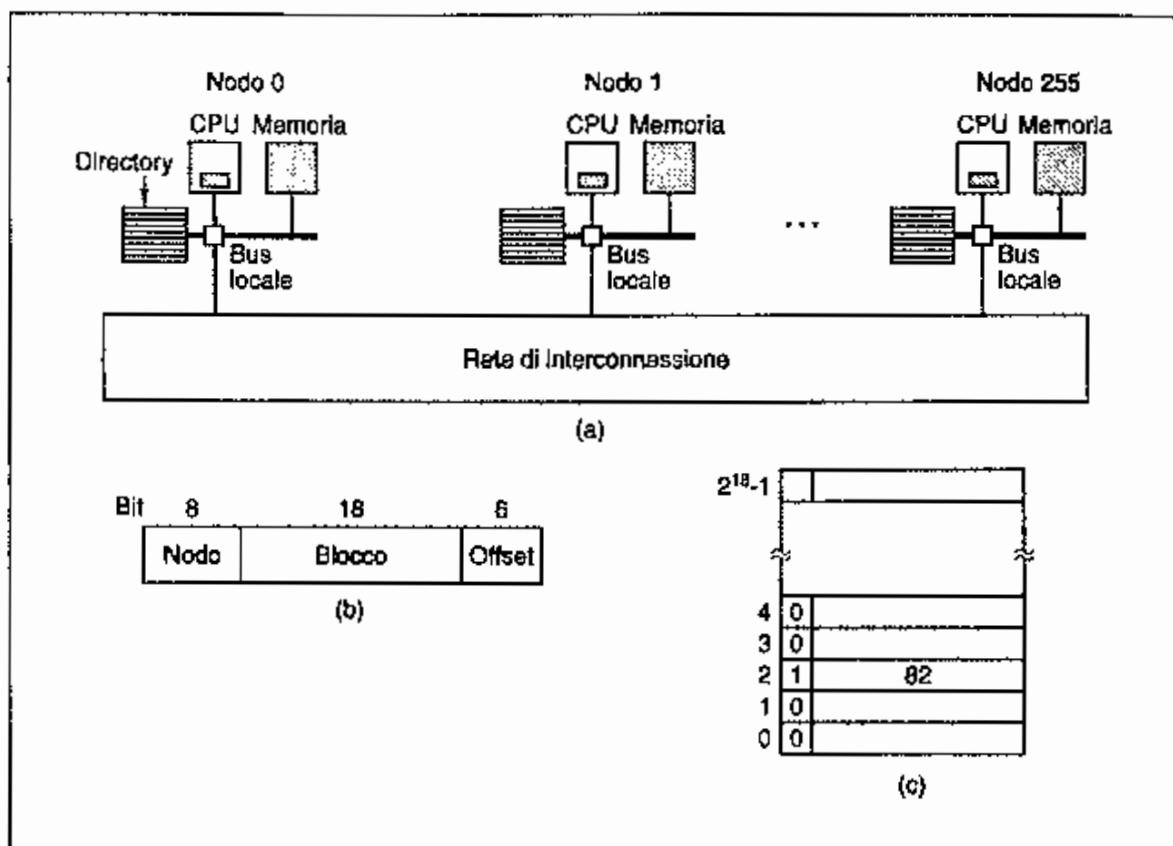


Figura 8.6 (a) Un multiprocessore basato su directory. (b) Divisione di un indirizzo di memoria in 32 campi. (c) La directory al nodo 36.

allocata fra i nodi, con gli indirizzi 0-16M nel nodo 0, 16M-32M nel nodo 1, e così via. I nodi sono connessi attraverso una rete di interconnessione, come mostrato nella Figura 8.6 (a); ciascuno mantiene anche le entry della directory per le  $2^{18}$  linee di cache da 64 byte, che formano i suoi  $2^{24}$  byte di memoria. Per il momento, supporremo che una linea possa essere mantenuta al massimo in una cache.

Per vedere come funziona la directory, consideriamo un'istruzione LOAD dalla CPU 20, che faccia riferimento a una linea della cache. Per prima cosa, la CPU spedisce l'istruzione alla propria MMU, che la trasforma in un indirizzo fisico, ad esempio 0 x 24000108, e lo suddivide nelle tre parti presentate nella Figura 8.6 (b). In decimale, le tre parti sono il nodo 36, la linea 4 e l'offset 8. La MMU si accorge che la parola di memoria riferita appartiene al nodo 36, non al nodo 20, e spedisce una richiesta attraverso la rete di interconnessione alla linea del nodo 36, chiedendo se la sua linea 4 sia nella cache, e in caso affermativo, dove sia.

Quando la richiesta arriva al nodo 36 attraverso la rete di interconnessione, è instradata all'hardware della directory, che ricerca l'indice nella sua tabella di  $2^{18}$  entrate, una per ciascuna delle linee di cache, ed estrae l'entry 4. Dalla Figura 8.6 (c) si vede che la linea non è nella cache, e così l'hardware estrae la linea 4 dalla RAM locale, la spedisce al nodo 20 e aggiorna la entry 4 della directory, per indicare che la linea si trova ora nella cache al nodo 20.

Si consideri ora una seconda richiesta relativa questa volta alla linea 2 del nodo 36: dalla Figura 8.6 (c) si vede che questa linea è nella cache al nodo 82. A questo punto, l'hardware potrebbe aggiornare l'entry 2 della directory, memorizzando il fatto che la linea si trova ora al nodo 20, quindi mandare un messaggio al nodo 82 istruendolo a inviare la linea al

nodo 20 e ad invalidare la propria cache. Si noti che anche in un cosiddetto "multiprocessore a memoria condivisa" c'è un intenso scambio di messaggi nascosti.

Già che ci siamo, calcoliamo brevemente quanta memoria è stata allocata per le directory: ciascun nodo ha 16 MB di RAM e  $2^{18}$  entry da 9 bit, per tenere traccia della propria RAM, quindi, l'overhead per la directory ammonta a circa  $9 \times 2^{18}$  bit diviso per 16MB, circa lo 1,76%, che è generalmente accettabile (benché si tratti di memoria ad alta velocità che fa salire il costo). Anche con linee di cache da 32 byte, l'overhead sarebbe solo del 4%, e con linee da 128 byte, sarebbe inferiore allo 1%.

Un'ovvia limitazione di questa architettura consiste nel fatto che una linea può essere nella cache di un solo nodo; per consentire alle linee di essere nella cache di qualunque nodo, avremmo bisogno di un inodo per localizzarle, ad esempio per invalidarle o aggiornarle in presenza di un'operazione di scrittura. Esistono varie possibilità per effettuare operazioni sulla cache in diversi nodi nello stesso tempo, ma il loro esame esula dagli intenti di questo libro.

### 8.1.2 Tipi di sistema operativo multiprocessore

Passiamo ora dall'hardware al software dei multiprocessori, in particolare ai sistemi operativi per multiprocessori. Esistono diverse organizzazioni, e qui di seguito ne presenteremo in esame tre.

#### Ciascuna CPU ha il proprio sistema operativo

Il modo più semplice per organizzare un sistema operativo multiprocessore consiste nel dividere staticamente la memoria in tante partizioni quante sono le CPU, fornendo a ciascuna CPU la propria memoria privata, e la propria copia del sistema operativo: in effetti, le  $n$  CPU operano allora come  $n$  computer indipendenti. Un'ovvia ottimizzazione è permettere a tutte le CPU di condividere il codice del sistema operativo e fare copie private solo dei dati, come vediamo nella Figura 8.7.

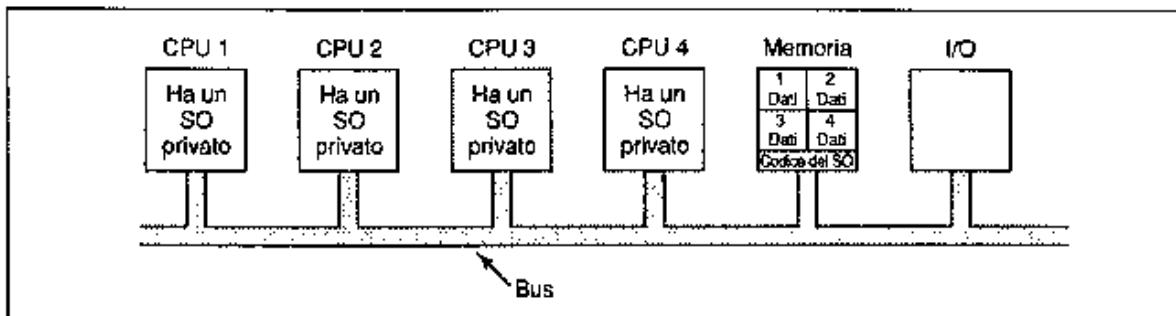
Questo schema è, in effetti, migliore rispetto a  $n$  computer separati, poiché permette a tutte le macchine di condividere un insieme di dischi e altri dispositivi di I/O, oltre ad una condivisione flessibile della memoria. Ad esempio, dovendo eseguire un programma di dimensioni insolitamente grandi, può essere allocata una porzione di memoria più estesa ad una delle CPU per tutta la durata di quel programma. Inoltre, i processi comunicano efficientemente tra loro, ad esempio un produttore può scrivere dati in memoria, e un consumatore può leggerli dallo stesso posto. Dal punto di vista del sistema operativo, tuttavia, il fatto che ciascuna CPU abbia il proprio sistema operativo, è piuttosto primitivo.

È opportuno ricordare esplicitamente quattro aspetti di questa architettura, che potrebbero non essere ovvi. In primo luogo, quando un processo effettua una chiamata di sistema, essa è intercettata e gestita dalla propria CPU, utilizzando le strutture dati nelle tabelle del proprio sistema operativo.

In secondo luogo, poiché ciascun sistema operativo ha le sue proprie tabelle, ha anche il proprio insieme di processi, che schedula per conto proprio. Non c'è condivisione di processi, e se un utente si connette alla CPU 1, tutti i suoi processi sono eseguiti su quella CPU. Può perciò accadere che la CPU 1 sia inattiva, mentre la CPU 2 sia carica di lavoro.

Terzo aspetto: non c'è condivisione di pagine. È possibile che la CPU 1 abbia pagine disponibili, mentre la CPU 2 continua ad utilizzare pagine, e non c'è modo, per la CPU 2, di ottenere pagine dalla CPU 1, poiché l'allocazione di memoria è fissa.

Il quarto e peggiore: se il sistema operativo mantiene una cache di blocchi del disco recentemente usati, ciascun sistema operativo lo fa indipendentemente dagli altri, perciò,



**Figura 8.7** Partizionamento della memoria del multiprocessore fra quattro CPU, con condivisione di una copia singola del codice del sistema operativo. I box etichettati con Dati sono i dati privati del sistema operativo per ciascuna CPU.

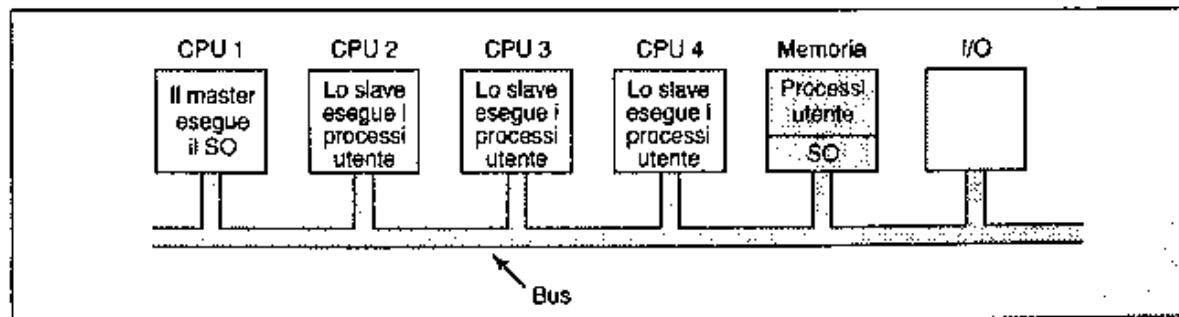
è possibile che un certo blocco del disco sia presente e "sporco" in diverse cache contemporaneamente, portando a risultati inconsistenti. Il solo modo per aggirare questo problema, è l'eliminazione delle cache: farlo non è difficile, ma diminuisce di molto le prestazioni. Per queste ragioni, tale modello è ormai desueto, benché sia stato impiegato nella fase iniziale dei sistemi multiprocessori, quando l'obiettivo era trasferire il più alla svelta possibile i sistemi operativi esistenti ai nuovi sistemi multiprocessore.

### Multiprocessori Master-Slave

Un secondo modello, è quello della Figura 8.8: qui il sistema operativo e le sue tabelle sono presenti sulla CPU 1, e non sulle altre; quindi tutte le chiamate di sistema sono reindirizzate alla CPU 1 perché siano ivi elaborate. La CPU 1 può anche eseguire processi utente, se rimane libero tempo di CPU. Questo modello è chiamato master-slave, poiché la CPU 1 è il 'padrone' (master), e tutte le altre sono gli "schiavi" (slave).

Il modello master-slave risolve la maggior parte dei problemi presenti nel modello precedente: c'è una sola struttura dati (ad esempio, una lista o un insieme di liste di priorità) che tiene traccia dei processi pronti, e quando una CPU è inattiva, richiede al sistema operativo un processo, che le viene assegnato; pertanto, non accade mai che una CPU sia inattiva, mentre un'altra è sovraccarica. Allo stesso modo, le pagine sono allocate dinamicamente fra tutti i processi, ed esiste una sola cache, cosicché non si verificano mai inconsistenze.

Il problema è che, con molte CPU, il master diventa un collo di bottiglia, perché deve gestire tutte le chiamate di sistema, per tutte le CPU. Se ad esempio il 10% del tempo passa nella gestione delle chiamate di sistema, dieci CPU saturano il master, e con venti CPU il sistema è sovraccarico; quindi, questo modello è semplice e utilizzabile per piccoli sistemi multiprocessori, ma non per quelli più grandi.



**Figura 8.8** Un modello multiprocessore master-slave.

## Multiprocessori simmetrici

Il nostro terzo modello, gli **SMP** (Symmetric MultiProcessor), elimina questa asimmetria: in memoria è presente una copia del sistema operativo, ma ogni CPU può eseguirlo. La CPU che ha effettuato la chiamata di sistema, effettua una trap al kernel, ed elabora la chiamata di sistema. Il modello SMP è illustrato nella Figura 8.9.

Questo modello bilancia i processi e la memoria dinamicamente, poiché c'è solo un insieme di tabelle del sistema operativo. Elimina anche il collo di bottiglia della CPU master, perché non esiste un master, tuttavia presenta anch'esso dei problemi; in particolare, se due o più CPU stanno eseguendo codice del sistema operativo in contemporanea, avverrà un disastro: basta pensare a due CPU che scelgano simultaneamente di eseguire lo stesso processo, o reclamino la stessa pagina di memoria libera. Il modo più semplice per evitare questi problemi, è associare un mutex (cioè un lock) al sistema operativo, rendendo l'intero sistema una grande regione critica. Quando una CPU vuole eseguire codice del sistema operativo, deve prima acquisire il mutex; se il mutex è bloccato, la CPU aspetta. Così, tutte le CPU possono eseguire il sistema operativo, ma solo una alla volta.

Questo modello funziona, ma con le stesse riserve del modello master-slave: supponiamo ancora che il 10% del tempo di esecuzione sia impiegato all'interno del sistema operativo, quindi con venti CPU, si formeranno lunghe code di CPU in attesa di accedere al sistema operativo. Per fortuna, è facile risolvere il problema: molte parti del sistema operativo sono indipendenti dalle altre. Ad esempio, non ci sono problemi quando una CPU esegue lo scheduler, un'altra sta gestendo una chiamata al file system, ed una terza elabora un fault di pagina.

Questa osservazione conduce a suddividere il sistema operativo in regioni critiche indipendenti, che non interagiscono l'una con l'altra. Ciascuna regione critica è protetta dal proprio mutex, e solo una CPU alla volta può eseguirla; così è possibile ottenere un parallelismo molto maggiore. Può tuttavia accadere che alcune tabelle, come quella dei processi, siano usate da diverse regioni critiche: ad esempio, la tabella dei processi è necessaria per la schedulazione, ma anche per la chiamata di sistema fork e per la gestione dei segnali. Ciascuna tabella che può essere usata da diverse regioni critiche, ha bisogno del proprio mutex, così ciascuna regione critica può essere eseguita da una sola CPU alla volta, e solo una CPU alla volta può accedere a ciascuna tabella critica.

La maggior parte dei moderni sistemi multiprocessore utilizza questa organizzazione. La parte più complessa riguardo alla scrittura del sistema operativo per una tale macchina, non consiste nel fatto che il codice effettivo sia così diverso da quello di un sistema operativo normale, perché non lo è. La parte più difficile è suddividerlo in regioni critiche che si possano eseguire concorrentemente da diverse CPU senza interferenze reciproche, neppure in modi sottili e indiretti. Inoltre, ogni tabella usata da due o più regio-

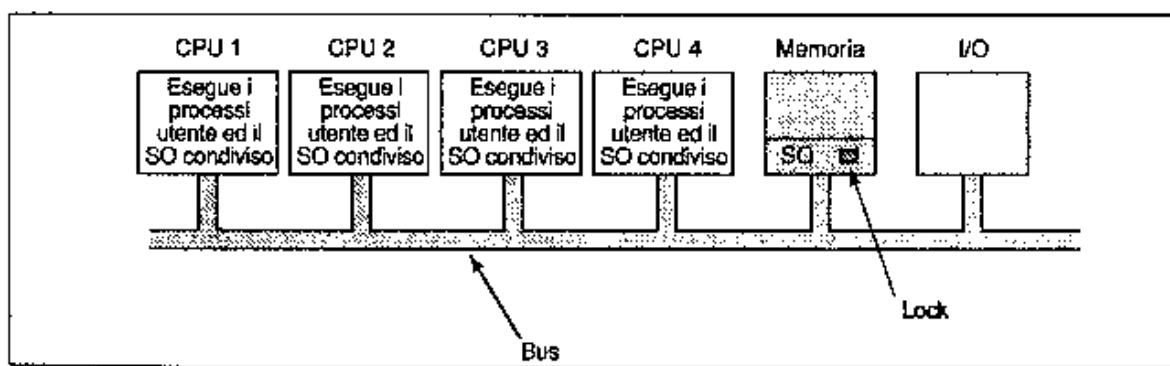


Figura 8.9 Il modello multiprocessore SMP.

ni critiche deve essere protetta separatamente da un mutex, e tutto il codice che utilizza la tabella deve usare il mutex correttamente.

Bisogna inoltre evitare i deadlock. Se due regioni critiche hanno bisogno entrambe della tabella A e B, e una di esse richiede A per prima, e l'altra richiede B per prima, prima o poi avverrà un deadlock, e nessuno saprà perché. In teoria, si potrebbero assegnare valori interi a tutte le tabelle, e richiedere che tutte le regioni critiche acquisiscano le tabelle in ordine crescente. Questa strategia evita i deadlock, ma richiede che il programmatore pensi con molta attenzione a quali tabelle siano necessarie per ciascuna regione critica per fare le richieste in ordine corretto.

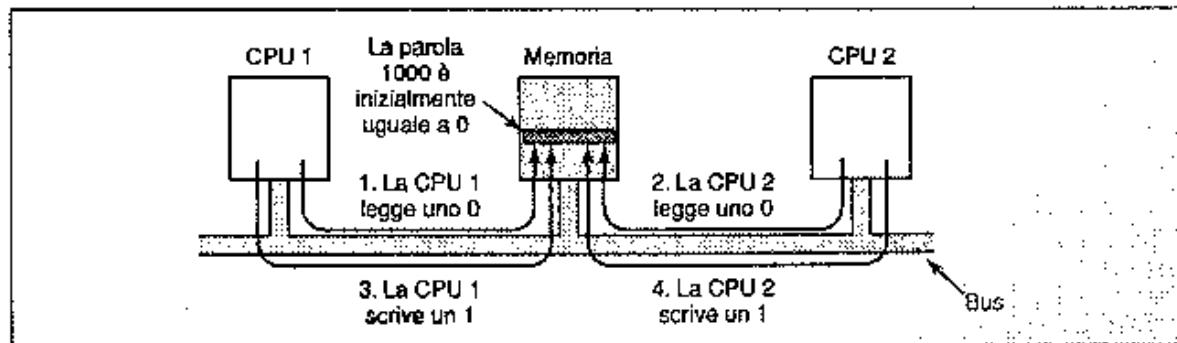
Man mano che il codice cambia nel tempo, una regione critica può aver bisogno di una nuova tabella che prima non le era necessaria. Se subentra un nuovo programmatore che non comprende appieno la logica del sistema, sarà tentato di porre un mutex sulla tabella di cui ha bisogno, e rilasciarlo quando la tabella non serve più. Per quanto ragionevole ciò possa apparire, può portare a deadlock, che l'utente percepirà come congelamento del sistema. Rendere il sistema corretto non è facile, e mantenerlo tale nel corso degli anni cambiando programmatore, è veramente difficile.

### 8.1.3 Sincronizzazione dei multiprocessori

Le CPU di un multiprocessore hanno bisogno di essere sincronizzate di frequente; abbiamo appena considerato il caso in cui le regioni critiche del kernel e le tabelle devono essere protette da un mutex. Guardiamo ora più da vicino come funziona effettivamente questa sincronizzazione: è tutt'altro che banale come vedremo presto.

In primo luogo, occorrono appropriate primitive per la sincronizzazione: se un processo su un monoprocessoresso effettua una chiamata di sistema che richieda l'accesso ad una tabella critica del kernel, il codice del kernel può disabilitare le interruzioni prima di toccare la tabella. Può quindi fare il proprio lavoro, sapendo che potrà finire senza che un altro processo s'infrustoli e acceda alla tabella prima che abbia finito. Su un multiprocessore, la disabilitazione delle interruzioni riguarda solo la CPU che la effettua, le altre CPU continuano l'esecuzione, e possono ancora accedere alla tabella critica. Di conseguenza, tutte le CPU devono usare e rispettare un protocollo mutex appropriato, per garantire che la mutua esclusione funzioni.

Il cuore di ogni protocollo mutex consiste in un'istruzione che permette di leggere e scrivere una parola di memoria in una sola operazione indivisibile. Abbiamo visto come utilizzare l'istruzione TSL (Test and Set Lock) nella Figura 2.22 per implementare le regioni critiche: essa legge una parola di memoria e la memorizza in un registro; nello stesso tempo, scrive un 1 (o un valore diverso da 0), nella parola di memoria. Naturalmente, per



**Figura 8.10** L'istruzione TSL può fallire se non è possibile bloccare il bus. Questi quattro passi mostrano una sequenza di eventi che conduce al fallimento della mutua esclusione.

effettuare la lettura e la scrittura in memoria, utilizza due cicli di bus separati. In un monoprocessoresso, se l'istruzione non può essere spezzata a metà dell'esecuzione, TSL funziona sempre secondo le attese.

Pensiamo ora a cosa potrebbe accadere su un multiprocessore. Nella Figura 8.10, consideriamo la peggior successione possibile di eventi temporali: la parola di memoria 1000, utilizzata come lock, è inizializzata a 0. Al passo 1, la CPU 1 legge la parola e ottiene uno 0. Al passo 2, prima che la CPU possa riscrivere la parola con 1, la CPU 2 legge a sua volta la parola, ottenendo uno 0. Al passo 3, la CPU 1 scrive un 1 nella parola, e al passo 4 anche la CPU 2 scrive un 1 nella parola. Entrambe le CPU hanno ottenuto uno 0 dall'istruzione TSL, e così entrambe ora hanno accesso alla regione critica, e la mutua esclusione fallisce.

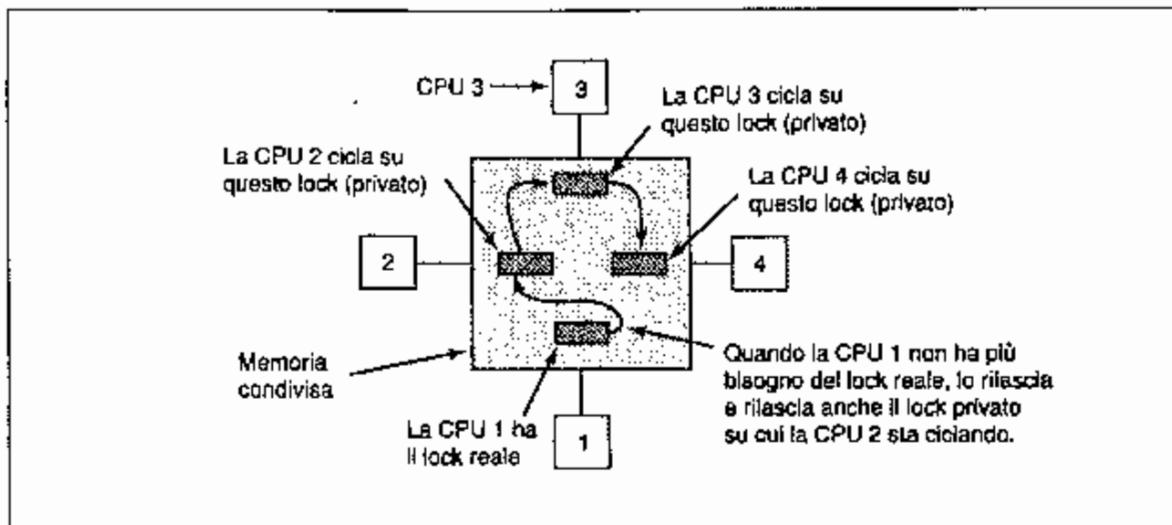
Per ovviare a questo problema, l'istruzione TSL deve prima bloccare il bus, in modo da evitare che altre CPU accedano ad esso, poi effettua entrambi gli accessi alla memoria, e sblocca il bus. Il lock del bus è effettuato richiedendolo al bus stesso, per mezzo del protocollo ordinario di richiesta, quindi ponendo a livello logico alto (cioè a livello logico 1) una linea speciale del bus, fino al completamento di entrambi i cicli; quando tale linea speciale è a livello logico alto, nessun'altra CPU può accedere al bus. È possibile implementare l'istruzione solo su un bus provvisto delle linee necessarie e del protocollo (hardware) per utilizzarle; i bus moderni hanno questa possibilità, ma su quelli precedenti, che ne sono privi, non è possibile implementare correttamente TSL. Per questa ragione è stato inventato il protocollo di Peterson, che permette la sincronizzazione interamente a livello software (Peterson 1981).

Se TSL è implementata e usata correttamente, la mutua esclusione funziona sempre; questo metodo di mutua esclusione utilizza uno spin lock, perché la CPU richiedente si impegnà in un loop stretto per testare il lock più rapidamente possibile, il che non solo spreca il tempo della CPU richiedente, ma anche pone un carico massiccio sul bus o sulla memoria, e rallenta tutte le altre CPU che cercano di svolgere il loro lavoro.

Potrebbe sembrare che la cache elimini il problema della contesa del bus, ma non è così: in teoria, una volta che la CPU richiedente ha letto la parola di lock, dovrebbe averne una copia nella sua cache, e finché nessun'altra CPU tenta di utilizzare il lock, la CPU richiedente dovrebbe poterlo ottenere dalla propria cache. Quando la CPU che possiede il lock scrive un 1 sul lock stesso, e lo rilascia, il protocollo di cache rende automaticamente non valide tutte le sue copie nelle cache remote, richiedendo che il valore corretto sia nuovamente prelevato.

Le cache, però, operano a blocchi di 32 o 64 byte. Di solito, la CPU che mantiene il lock ha bisogno delle parole che lo circondano e, poiché l'istruzione TSL è una scrittura (in quanto modifica il lock), ha bisogno di accesso esclusivo al blocco della cache che contiene il lock. Quindi ogni TSL rende non valido il blocco della cache in cui è presente il lock, e preleva una copia privata ed esclusiva per la CPU richiedente. Di conseguenza, l'intero blocco della cache che contiene il lock, fa la spola fra il possessore e il richiedente del lock, generando un traffico sul bus anche maggiore di quello che si avrebbe con le letture della sola parola di lock.

Se potessimo sbarazzarci di tutte queste scritture indotte da TSL da parte della CPU richiedente, potremmo ridurre apprezzabilmente il thrashing della cache: questo è possibile, facendo in modo che la CPU richiedente effettui prima una semplice lettura, per vedere se il lock è libero, e, in caso affermativo, effettui una TSL per acquisirlo. Il risultato di questa piccola correzione è che la maggior parte delle operazioni sono ora di lettura, piuttosto che di scrittura; se la CPU che mantiene il lock sta solo leggendo le variabili nel blocco della cache che contiene il lock, esse possono avere una copia ciascuna del blocco della cache, in modalità di sola lettura condivisa, eliminando tutti i trasferimenti dei blocchi della cache. Quando il lock finalmente viene liberato, il pos-



**Figura 8.11** Uso di lock multipli per evitare il trashing della cache.

sessore effettua una scrittura che richiede accesso esclusivo, invalidando così tutte le altre copie nelle cache remote. Alla successiva lettura da parte della CPU richiedente, il blocco della cache sarà ricaricato. Si noti che se due o più CPU sono in contesa per il medesimo lock, può accadere che entrambe si accorgano che è libero simultaneamente, ed entrambe facciano, nello stesso momento, una TSL per acquisirlo. Solo una avrà successo, e non si avranno corse critiche a questo livello, perché l'acquisizione reale è effettuata dall'istruzione TSL, che è atomica. Accorgersi che il lock è libero e quindi provare ad ottenerlo immediatamente con una TSL, non garantisce che lo si ottenga: qualcun altro potrebbe riuscirci prima.

Un altro modo per ridurre il traffico del bus consiste nell'utilizzo dell'algoritmo Ethernet di backoff esponenziale binario (Anderson, 1990). Invece di un polling continuo, come nella Figura 2.22, si può inserire un loop di ritardo fra i cicli di poll. Inizialmente il ritardo è di una istruzione, se il lock è ancora occupato, il ritardo è raddoppiato a due istruzioni, quindi a quattro, e così via fino a un qualche massimo. Un massimo relativamente basso fornisce una risposta rapida quando il lock è rilasciato, ma spreca più cicli di bus in presenza di trashing della cache. Un massimo relativamente elevato riduce il trashing della cache al prezzo di una notifica non immediata della disponibilità del lock. L'algoritmo di backoff binario esponenziale può utilizzare o meno la lettura prima dell'istruzione TSL.

Un'idea ancora migliore consiste nel dare a ciascuna CPU che voglia acquisire il mutex la sua propria variabile di lock da testare, come illustrato nella Figura 8.11 (Mellor-Crummery e Scott, 1991); tale variabile dovrebbe risiedere in un blocco della cache che altrimenti rimarrebbe inutilizzato, per evitare conflitti. L'algoritmo funziona facendo in modo che una CPU che fallisce ad acquisire il lock allochi una variabile di lock e includa se stessa in coda ad una lista di CPU che attendono il lock. Quando la CPU che detiene il lock esce dalla regione critica, libera il lock privato che la prima CPU della lista sta testando (nella propria cache). Questa CPU entra allora nella regione critica, e, quando ha finito, libera il lock del suo successore e così via. Benché il protocollo sia abbastanza complicato (per evitare che due CPU si accodino simultaneamente alla lista), è efficiente ed immune da starvation. Per maggiori dettagli invitiamo il lettore a consultare il relativo articolo.

## Spinning e switching

Abbiamo appena ipotizzato che una CPU che abbia bisogno di un mutex bloccato rimanga in sua attesa, effettuando il polling continuamente, o a intermittenza, oppure accodandosi ad una lista di CPU in attesa. In molti casi, per la CPU richiedente non ci sono vere alternative all'attesa: ad esempio, se una CPU è inattiva, ha bisogno di accedere alla lista condivisa dei pronti per scegliere un processo da eseguire. Se la lista dei pronti ha un lock, la CPU non può semplicemente decidere di sospendere ciò che sta facendo ed eseguire un altro processo, perché per farlo dovrebbe accedere alla lista dei pronti: deve aspettare, finché può acquisire la lista dei pronti.

Comunque, in altri casi è possibile scegliere, ad esempio, se un thread di una CPU ha bisogno di accedere alla cache del file system, e se tale cache ha correntemente un lock, la CPU può decidere di passare ad un thread differente, invece di aspettare. Il problema di scegliere se ciclare (spinning) oppure effettuare un cambio di thread (switching), è stato oggetto di molta ricerca, parte della quale verrà esaminata qui di seguito. Si noti che ciò non capita su un monoprocessore, perché spinning è privo di senso se nessun'altra CPU può liberare il lock. Se un thread cerca di acquisire un lock e fallisce, è sempre bloccato per dare al possessore del lock la possibilità di andare in esecuzione, e rilasciare il lock stesso.

Supponendo che lo spinning e il cambiamento di thread siano entrambi ammissibili, il trade off è questo: lo spinning spreca direttamente cicli di CPU; testare un lock ripetutamente non è produttivo. Anche lo switching in ogni modo spreca cicli di CPU, poiché occorre salvare lo stato corrente del thread, acquisire il lock della lista dei pronti, selezionare un thread, poi bisogna caricare il suo stato e infine mandarlo in esecuzione. Inoltre, la cache della CPU conterrà soltanto blocchi sbagliati, e così avverranno una serie di costose cache miss, appena il nuovo thread inizierà l'esecuzione; sono anche probabili fault di TLB. Prima o poi, si avrà un ritorno al thread originale, con conseguenti ulteriori cache miss: i cicli spesi per effettuare questi due cambiamenti di contesto, più tutte le cache miss, sono sprecati.

Sapendo che i mutex sono mantenuti generalmente per 50 µs, se serve 1 ms per cambiare il thread corrente, e 1 ms per ritornarvi dopo, è più efficiente ciclare sul mutex; se invece il mutex è tenuto mediamente per 10 ms, vale la pena di effettuare due cambi di contesto. Il problema è che le regioni critiche possono variare considerevolmente nella loro durata, quindi quale approccio è il migliore?

Una possibilità è ciclare sempre, una seconda è di effettuare sempre il cambiamento di contesto; ma un terzo schema consiste nel decidere indipendentemente, ogni volta che s'incontra un mutex con lock. Al momento della decisione, si ignora se sia meglio ciclare o cambiare contesto, ma per un dato sistema è possibile prendere nota di tutte le attività e analizzarle successivamente off line. Quindi, a posteriori è possibile individuare la decisione migliore, e quanto tempo è stato sprecato nel caso migliore; tale algoritmo scritto col senso di poi diviene un benchmark, rispetto al quale si possono misurare gli algoritmi reali.

Parecchi ricercatori si sono occupati di questo problema (Kartin et al., 1989; Karlin et al., 1991; Ousterhout, 1982). La maggior parte degli studi utilizza un modello in cui un thread che fallisce ad acquisire un mutex cicla per un certo periodo di tempo, e se supera un certo limite, viene effettuato un cambio di contesto. In qualche caso il limite è fissato, ad esempio l'overhead noto per passare ad un altro thread e quindi tornare indietro. In altri casi è dinamico, dipendente dalla storia del mutex che si stava aspettando.

I migliori risultati si ottengono quando il sistema tiene traccia delle ultime volte in cui si sono verificati cicli d'attesa, e suppone che quello attuale sia simile ai precedenti. Ad esempio, supponendo di nuovo un tempo di cambiamento di contesto di 1 ms, un thread

ciclerebbe per un massimo di 2ms, osservando però quanto realmente ha ciclato. Se fallisce ad acquisire un lock e vede che ha aspettato in media 200 µs nelle precedenti tre volte, dovrebbe ciclare per 2 ms prima di cambiare contesto; se invece vede che ha girato per 2 ms in ciascuno dei tentativi precedenti, dovrebbe cambiare contesto immediatamente e non ciclare. Si possono trovare più dettagli in Karlin et al., 1991.

### 8.1.4 Schedulazione dei multiprocessori

In un monoprocesso, la schedulazione è monodimensionale. La sola domanda cui si deve rispondere (più volte) è: "qual è il prossimo processo da eseguire?" In un multiprocesso, la schedulazione è bidimensionale: lo scheduler deve decidere quale processo eseguire, e su quale CPU eseguirlo, e questa seconda dimensione complica notevolmente la schedulazione dei multiprocessori.

Un altro fattore di complicazione è che in alcuni sistemi tutti i processi non sono correlati, mentre in altri lo sono. Un esempio della prima situazione si ha nei sistemi time-sharing dove utenti indipendenti iniziano processi indipendenti: i processi non sono correlati e ciascuno può essere schedulato senza considerare gli altri.

La seconda situazione si verifica regolarmente negli ambienti di sviluppo dei programmi: i grandi sistemi spesso si compongono di alcuni file header che contengono macro, definizioni di tipo e dichiarazioni di variabili utilizzate dai file di codice. Quando si cambia un file header, occorre ricompilare tutti i file di codice che lo includono. Il programma *make* è comunemente usato per gestire l'ambiente di sviluppo: quando lo si lancia, *make* inizia la compilazione solo di quei file di codice da ricompilare, tenendo conto dei cambiamenti ai file header o di codice, mentre i file oggetto ancora validi non sono rigenerati.

La versione originale di *make* viene eseguita sequenzialmente, ma le versioni più recenti, progettate per multiprocessori, possono iniziare tutte le compilazioni contemporaneamente. Se sono necessarie dieci compilazioni, non ha senso schedularne nove rapidamente e lasciare l'ultima molto indietro, poiché l'utente non vedrà il lavoro completo, sino alla fine dell'ultima compilazione. In questo caso ha senso considerare i processi come un gruppo, e tenerne conto durante la schedulazione.

#### Timesharing

Per prima cosa, esamineremo la schedulazione di processi indipendenti, quindi vedremo come schedulare processi correlati. Il più semplice algoritmo di schedulazione, per trattare processi (o thread) non correlati, ha una sola struttura dati di sistema per i processi pronti, ad esempio una lista, ma più probabilmente un insieme di liste per processi a differente priorità, come vediamo nella Figura 8.12 (a). Qui le sedici CPU sono tutte al momento occupate, e un insieme di quattordici processi con priorità sta aspettando di andare in esecuzione. La prima CPU a finire il proprio lavoro corrente (o ad avere il proprio processo bloccato), è la CPU 4, che poi effettua il lock delle code di schedulazione e seleziona il processo a maggior priorità, A, come mostrato nella Figura 8.12 (b); successivamente, la CPU 12 diventa inattiva, e sceglie il processo B, come si vede nella Figura 8.12 (c); se i processi sono completamente scorrelati, effettuare così la schedulazione è una scelta ragionevole.

La schedulazione con una sola struttura dati utilizzata da tutte le CPU, effettua il time-sharing delle CPU in gran parte come se ci trovassimo in un sistema monoprocesso, fornendo anche un bilanciamento automatico del carico perché non può mai accadere che una CPU sia inattiva, e le altre sovraccaricate. Due svantaggi di questo approccio sono la

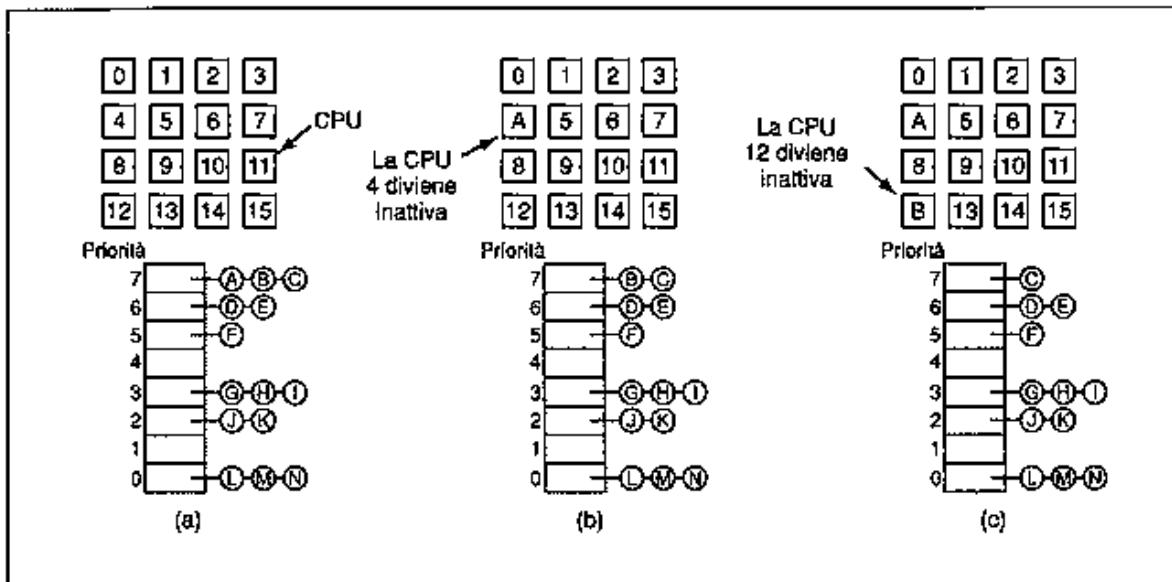


Figura 8.12 Schedulazione di un multiprocessore con una sola struttura dati.

contesa potenziale per la struttura dati di schedulazione, quando il numero delle CPU cresce, e (come al solito) l'overhead nell'effettuare un cambio di contesto, quando un processo si blocca per l'I/O.

È anche possibile che un cambio di contesto avvenga quando finisce il quanto di tempo di un processo, ma un multiprocessore ha alcune proprietà non presenti in un sistema monoprocesso. Si supponga che il processo detenga un lock gestito con cicli di attesa, cosa non insolita nei multiprocessori, come abbiamo visto prima; altre CPU in attesa del lock sprecano tempo ciclando finché quel processo è nuovamente schedulato, e rilascia il lock. Nei sistemi monoprocesso, tali tipi di lock si utilizzano raramente, cosicché se un processo è sospeso mentre detiene un mutex, e un secondo processo parte, e prova ad acquisirlo, sarà immediatamente bloccato, con poco spreco di tempo.

Per ovviare a quest'anomalia, alcuni sistemi utilizzano lo *smart scheduling*, in cui un processo, che acquisisce un lock gestito con cicli di attesa, utilizza un flag a livello di processo per evidenziare che detiene correntemente un lock (Zahorjan et al., 1991); quando rilascia il lock, azzera il flag. Lo schedulatore, a quel punto, non ferma un processo che detiene un lock, bensì gli fornisce un po' di tempo in più per completare la sua regione critica e rilasciare il lock.

Un altro problema inherente la schedulazione è che, mentre tutte le CPU sono uguali, alcune sono più uguali delle altre; in particolare, quando un processo A viene eseguito a lungo sulla CPU k, la cache della CPU k sarà piena di blocchi del processo A; se A torna di nuovo in esecuzione subito, può ottenere migliori prestazioni se verrà eseguito sulla CPU k, perché la cache di k può contenere ancora alcuni blocchi di A. I blocchi della cache precaricati incrementeranno l'hit rate della cache e quindi la velocità del processo. Inoltre, anche il TLB può contenere le pagine giuste, riducendo i relativi fault.

Alcuni multiprocessori tengono conto di questo effetto, e utilizzano la **schedulazione per affinità** (Vaswani e Zahorjan, 1991). L'idea base è sforzarsi di eseguire un processo sulla stessa CPU dove è stato eseguito l'ultima volta. Un modo per creare questa affinità è usare un **algoritmo di schedulazione a due livelli**: quando un processo è creato, viene assegnato ad una CPU, ad esempio a quella che al momento ha il minor carico, e tale assegnazione dei processi alla CPU costituisce il livello superiore dell'algoritmo. Come risultato, ciascuna CPU acquisisce la propria collezione di processi.

La schedulazione effettiva dei processi costituisce il livello inferiore dell'algoritmo, e viene effettuata da ciascuna CPU separatamente, utilizzando le priorità o altri mezzi. Cercando di mantenere un processo sulla stessa CPU, si massimizza l'affinità della cache; se però una CPU non ha processi da eseguire, può prenderli da un'altra, piuttosto che rimanere inattiva.

La schedulazione a due livelli comporta tre vantaggi. In primo luogo distribuisce il carico fra le CPU disponibili in modo abbastanza paritario; in secondo luogo, sfrutta l'affinità della cache, dove possibile; infine, fornendo a ciascuna CPU la propria lista dei pronti, si minimizza la contesa per le liste dei pronti, poiché i tentativi di utilizzare la lista dei pronti di un'altra CPU sono relativamente poco frequenti.

## Condivisione dello spazio

L'altro approccio generale alla schedulazione dei multiprocessori si può usare quando esiste una qualche correlazione fra i processi. Più sopra abbiamo ricordato l'esempio del *make* parallelo. Accade però di frequente che un singolo processo crei differenti thread che lavorano insieme. Ai nostri fini, un insieme di diversi processi correlati, oppure un processo che consista di diversi thread del kernel, sono fondamentalmente la stessa cosa. Ci riferiremo alle entità schedulabili come a thread, ma la trattazione è ugualmente valida per i processi. Schedulare diversi thread nello stesso tempo, per mezzo di diverse CPU, è chiamato **space sharing**, o **condivisione dello spazio**.

Il più semplice algoritmo di condivisione dello spazio funziona così. Si supponga di creare contemporaneamente un insieme di thread correlati; in tale momento, lo scheduler controlla se ci sono tante CPU libere quanti thread. Se esistono, assegna a ciascun thread la sua CPU dedicata (non multiprogrammata), e partono tutti; se non ci sono abbastanza CPU, non si fa partire nessun thread, finché non è disponibile un numero adeguato di CPU. Ciascun thread tiene la sua CPU sino al termine e, a quel punto, la CPU è rimessa nell'insieme delle CPU disponibili; se ad esempio un thread si blocca sull'I/O, continua a tenere la CPU, che riunirà inattiva finché il thread si risveglia. Quando appare il successivo gruppo di thread, si applica lo stesso algoritmo: in ogni istante, l'insieme delle CPU è suddiviso in un certo numero di partizioni, ciascuna delle quali esegue i thread di un processo. Nella Figura 8.13 abbiamo partizioni di dimensioni 4, 6, 8, e 12 CPU, con due CPU non assegnate; man mano che si procede, il numero e la dimensione delle partizioni cambierà secondo i processi presenti.

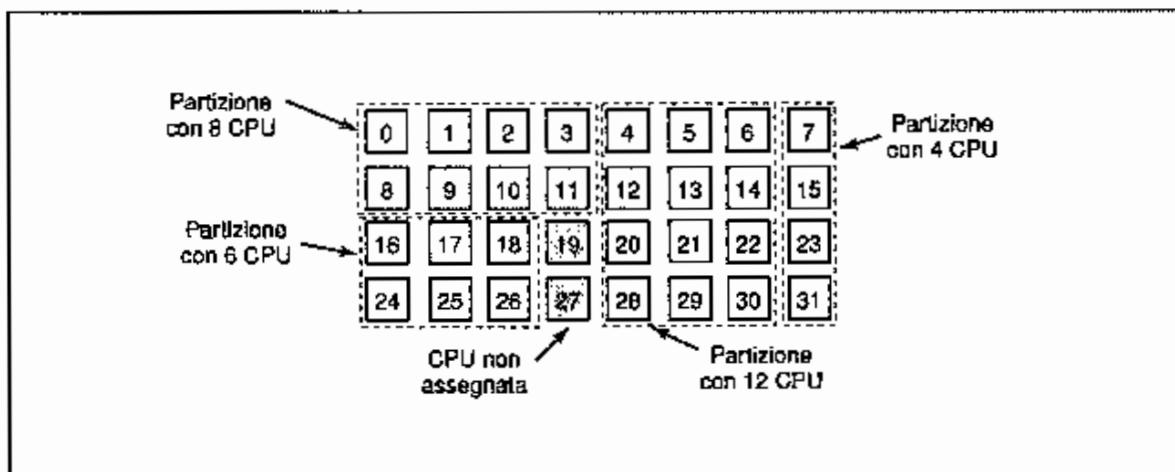


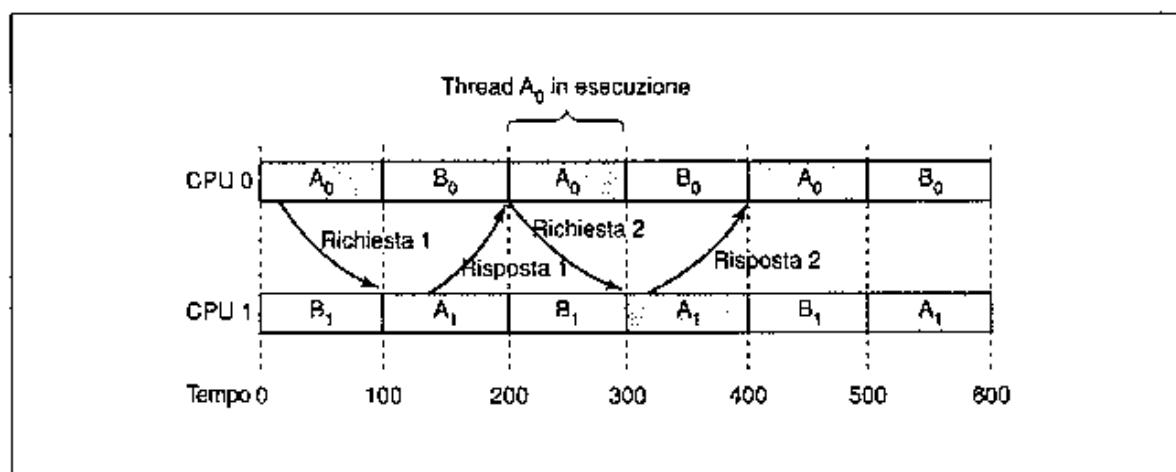
Figura 8.13 Un insieme di 32 CPU suddiviso in quattro partizioni, con due CPU disponibili.

Periodicamente, si devono prendere decisioni di schedulazione; in sistemi monoprocessoressi l'algoritmo shortest job first è usato per la schedulazione in batch; l'algoritmo analogo per un multiprocessore consiste nello scegliere il processo che ha bisogno del minor numero di cicli di CPU, cioè il processo il cui (numero-di-CPU  $\times$  tempo di esecuzione) è il più piccolo tra i candidati. Tuttavia, in pratica, questa informazione è raramente disponibile, e l'algoritmo è difficile da implementare; inoltre, gli studi hanno mostrato che, in pratica, è difficile superare le prestazioni dell'algoritmo primo-arrivato primo-servito (Krueger et al., 1994).

In questo semplice modello di partizione, un processo richiede un certo numero di CPU e, o le ottiene tutte, oppure deve aspettare finché siano disponibili. Un altro approccio riguarda i processi capaci di gestire attivamente il grado di parallelismo, per esempio con un server centrale che tenga traccia di quali processi siano presenti e vogliano andare in esecuzione, e delle loro richieste di CPU minime e massime (Tucker e Gupta, 1999). Periodicamente ciascuna CPU chiede al server centrale di quante CPU il server stesso disponga; in relazione alla disponibilità, aggiusta verso l'alto o verso il basso il numero dei processi o dei thread. Ad esempio, un Web server può avere 1, 2, 5, 10, 20 o quant'altre thread in esecuzione in parallelo; se al momento ha dieci thread, e se ha improvvisamente c'è una maggior richiesta di CPU e gli viene detto di scendere a cinque, quando i successivi cinque thread finiscono il loro lavoro corrente vengono fatti terminare, e non è loro assegnato nessun nuovo lavoro. Questo schema permette una variazione dinamica delle dimensioni della partizione, in modo che, rispetto al sistema fisso della Figura 8.13, corrisponda meglio al carico di lavoro corrente.

## Schedulazione gang

Un chiaro vantaggio della condivisione dello spazio è l'eliminazione della multiprogrammazione, che elimina l'overhead dovuto al cambiamento di contesto. Uno svantaggio ugualmente evidente è il tempo perduto quando una CPU si blocca, e non ha nulla da fare, finché non diviene nuovamente pronta. Di qui, la ricerca di algoritmi che tentano di schedulare spazio e tempo insieme, in particolare per processi che creano thread multipli, i quali di solito hanno bisogno di comunicare fra di loro. Per vedere il tipo di problemi che possono verificarsi quando i thread di un processo (o i processi di un job) sono schedulati indipendentemente, si consideri un sistema con i thread  $A_0$  ed  $A_1$  che



**Figura 8.14** Comunicazione fra due thread, appartenenti al processo A, che sono in esecuzione fuori fase.

appartengono al processo  $A$ , e i thread  $B_0$  e  $B_1$ , che appartengono al processo  $B$ ; i thread  $A_0$  e  $B_0$  sono in time sharing sulla CPU 0,  $A_1$  e  $B_1$  sulla CPU 1. I thread  $A_0$  e  $A_1$  hanno bisogno spesso di comunicare:  $A_0$  manda un messaggio ad  $A_1$ ,  $A_1$  manda poi una risposta ad  $A_0$ , seguita da un'altra sequenza uguale. Si supponga che, per caso,  $A_0$  e  $B_1$  partano prima, come vediamo nella Figura 8.14.

Nella porzione di tempo 0,  $A_0$  manda una richiesta ad  $A_1$ , ma  $A_1$  non la riceve fino alla porzione di tempo 1, che inizia dopo 100 ms.  $A_1$  spedisce immediatamente la risposta, ma  $A_0$  non la riceve finché non viene nuovamente eseguito, dopo 200 ms. Il risultato è una sequenza di richiesta-risposta ogni 200 ms, il che non è bene.

La soluzione a questo problema è fornita dalla **schedulazione-gang**, uno sviluppo della **co-schedulazione** (Ousterhout, 1982). La schedulazione gang è costituita di tre parti:

1. Gruppi di thread correlati sono schedulati come un'unità, una gang.
2. Tutti i membri di una gang sono in esecuzione simultaneamente, in differenti CPU in timesharing.
3. Tutti i membri di una gang iniziano e finiscono le loro porzioni di tempo insieme.

La schedulazione gang funziona perché tutte le CPU sono schedulate in sincronia: questo significa che il tempo è suddiviso in quanti discreti, come nella Figura 8.14. All'inizio di ciascun nuovo quanto, tutte le CPU sono di nuovo schedolate, e un nuovo thread inizia su ciascuna di esse. All'inizio del quanto seguente, avviene un altro evento di schedulazione; in mezzo, non si fa alcuna schedulazione, e se un thread si blocca, la sua CPU rimane inattiva sino alla fine del quanto.

Un esempio del funzionamento di questa schedulazione è presentato nella Figura 8.15. Abbiamo un multiprocessore con sei CPU, usate da cinque processi, da  $A$  ad  $E$ , per un totale di 24 thread pronti. Durante l'intervallo di tempo 0, sono schedolati ed eseguiti i thread da  $A_0$  ad  $A_4$  durante l'intervallo di tempo 1, i thread  $B_0$ ,  $B_1$ ,  $B_2$ ,  $C_0$ ,  $C_1$ ,  $C_2$ . Durante l'intervallo di tempo 2, sono mandati in esecuzione i cinque thread di  $D$  ed  $E_0$ . I rimanenti sei thread del processo  $E$ , sono eseguiti nell'intervallo di tempo 3. Quindi il ciclo si ripete, con l'intervallo di tempo 4, identico all'intervallo 0.

L'idea della schedulazione gang consiste nell'avere tutti i thread di un processo in esecuzione insieme, cosicché se uno di essi manda una richiesta ad un altro, quest'ultimo riceverà il messaggio e sarà in grado di rispondere quasi immediatamente. Nella Figura

| CPU                 |       |       |       |       |       |       |
|---------------------|-------|-------|-------|-------|-------|-------|
|                     | 0     | 1     | 2     | 3     | 4     | 5     |
| Intervallo di tempo | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 0                   | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 1                   | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 2                   | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |
| 3                   | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 4                   | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 5                   | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 6                   | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |
| 7                   |       |       |       |       |       |       |

Figura 8.15 Schedulazione gang.

8.15, poiché tutti i thread di *A* sono in esecuzione insieme durante un quanto, in tale intervallo di tempo possono mandare e ricevere un gran numero di messaggi, eliminando così il problema della Figura 8.14.

## 8.2 Multicomputer

I multiprocessori sono diffusi e interessanti, perché offrono un semplice modello di comunicazione: tutte le CPU condividono una memoria comune. I processi possono scrivere messaggi in memoria, letti poi da altri processi; la sincronizzazione si può realizzare con mutex, semafori, monitor ed altre tecniche ben consolidate; l'unico problema è che multiprocessori di grandi dimensioni sono difficili da costruire, e quindi costosi.

Per aggirare queste difficoltà, è stata fatta molta ricerca sui multicomputer, che sono CPU strettamente accoppiate, che non condividono memoria. Ciascuna ha la propria memoria, come si vede nella Figura 8.1 (b). Questi sistemi sono noti sotto molti nomi diversi, ad esempio cluster di computer e COWS (Clusters of Workstations).

I multicomputer sono facili da costruire perché la componente base è un normale PC, con l'aggiunta di una scheda di rete. Naturalmente, il segreto per ottenere buone prestazioni risiede nel progettare la rete di interconnessione e la scheda di interfaccia in modo conveniente: questo problema è decisamente simile a quello della costruzione della memoria condivisa in un multiprocessore, ma ora l'obiettivo è mandare messaggi in un tempo dell'ordine dei microsecondi, piuttosto che accedere alla memoria in un tempo dell'ordine dei nanosecondi, e così l'obiettivo è più semplice, facile ed economico da realizzare.

Nelle prossime sezioni, ci occuperemo brevemente dell'hardware dei multicomputer, specialmente dell'hardware di interconnessione; quindi passeremo al software, partendo dal software di comunicazione di basso livello, per poi esaminare quello di alto livello, e prendere in esame un modo per realizzare la memoria condivisa in sistemi che ne sono privi. Infine, passeremo alla schedulazione ed al bilanciamento del carico.

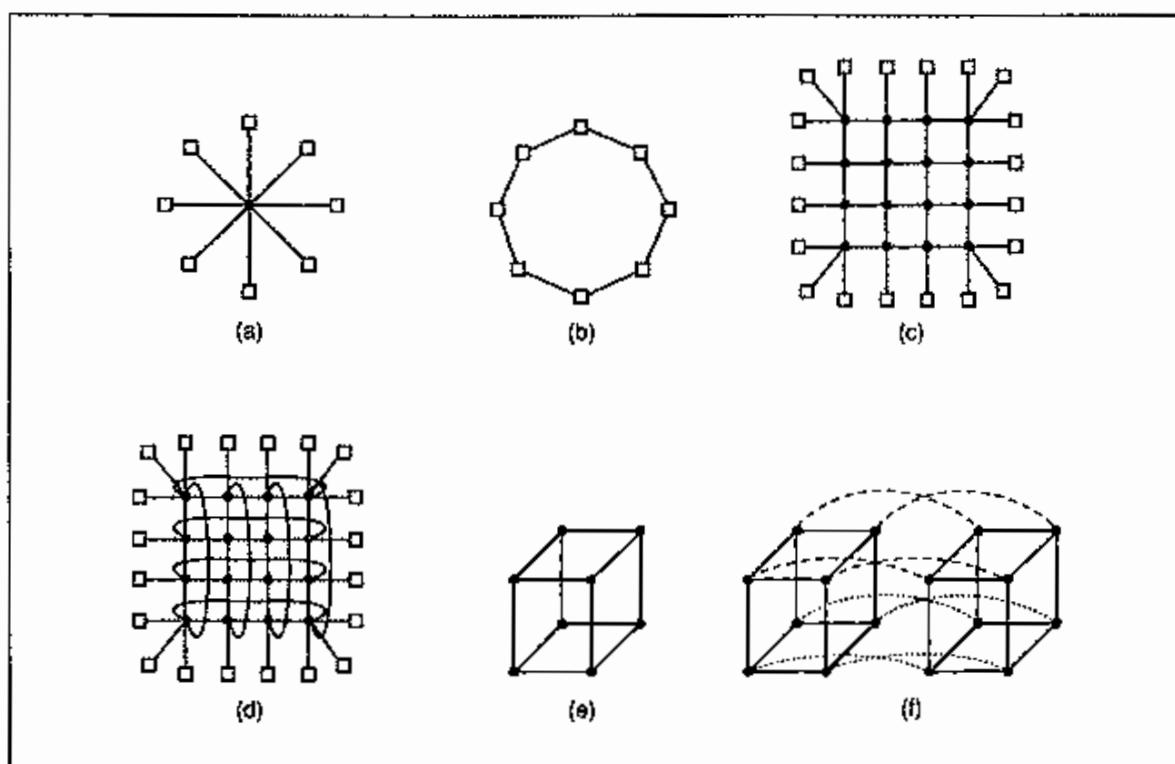
### 8.2.1 Hardware dei multicomputer

Il nodo base di un multicomputer si compone di una CPU, memoria, un'interfaccia di rete e, talvolta, un hard disk. Il nodo può essere sistemato in un case standard per PC, ma la scheda grafica, il monitor, la tastiera e il mouse sono quasi sempre assenti. In alcuni casi, il PC contiene una scheda multiprocessore a due o quattro vie, invece di una singola CPU, ma per semplicità supporremo che ciascun nodo possa avere una sola CPU. Spesso centinaia o anche migliaia di nodi sono collegati insieme per formare un multicomputer. Qui di seguito, vedremo in breve com'è organizzato questo hardware.

#### Tecnologia di interconnessione

Ciascun nodo ha un'interfaccia di rete con uno o due cavi (o fibre) uscenti da essa; tali cavi si connettono ad altri nodi, o a switch (moduli di connessione). In un piccolo sistema ci può essere uno switch cui tutti i nodi sono connessi, come nella topologia a stella della Figura 8.16 (a), utilizzata dalle moderne Ethernet switched.

Come alternativa all'architettura a singolo switch, i nodi possono formare un anello, con due cavi che escono dall'interfaccia di rete, uno che va al nodo di sinistra e l'altro al nodo di destra, come vediamo nella Figura 8.16 (b). In questa topologia non sono necessari switch.



**Figura 8.16** Varie topologie di interconnessione. (a) Uno switch singolo. (b) Un anello. (c) Una griglia. (d) Un toro doppio. (e) Un cubo. (f) Un ipercubo 4D.

La griglia o mesh di Figura 8.16 (c), è un'architettura bidimensionale, utilizzata in molti sistemi commerciali; è molto regolare e facilmente scalabile a grandi dimensioni. Ha un diametro corrispondente al più lungo cammino fra due nodi, che cresce solo con la radice quadrata del numero dei nodi. Una variante della griglia è il doppio toro della Figura 8.16 (b), che è una griglia con i lati connessi. Non solo è più resistente agli errori rispetto alla griglia, ma ha un diametro minore, poiché gli angoli opposti possono comunicare in soli due passi.

Il cubo della Figura 8.16 (e), è una topologia regolare, tridimensionale: abbiamo illustrato un cubo  $2 \times 2 \times 2$ , ma, in generale, potrebbe essere un cubo  $k \times k \times k$ . Nella Figura 8.16 (f) abbiamo un cubo a quattro dimensioni, costruito con due cubi tridimensionali con i nodi corrispondenti connessi. Potremmo costruire un cubo a cinque dimensioni, clonando la struttura della Figura 8.16 (f), e connettendo i nodi corrispondenti per formare un blocco di quattro cubi. Per arrivare a sei dimensioni potremmo replicare il blocco di quattro cubi e connettere i nodi corrispondenti, e così via. Un cubo n-dimensionale, formato così, è chiamato ipercubo: molti computer paralleli usano questa topologia perché il diametro cresce linearmente con la dimensione. In altre parole, il diametro è il logaritmo in base 2 del numero dei nodi, quindi, ad esempio, un ipercubo di dimensione 10 ha 1024 nodi, ma un diametro solo di 10, con eccellenti proprietà di ritardo. Si noti che, al contrario, 1024 nodi organizzati in una griglia  $32 \times 32$  hanno un diametro di 62, oltre sei volte maggiore di quello dell'ipercubo. Il prezzo per il diametro inferiore è che il fanout, e quindi il numero dei collegamenti (e il costo), è molto maggiore per l'ipercubo.

Nei multicomputer si utilizzano due tipi di schemi di connessione. Nel primo, ciascun messaggio prima è suddiviso (o dal software utente, o dall'interfaccia di rete) in

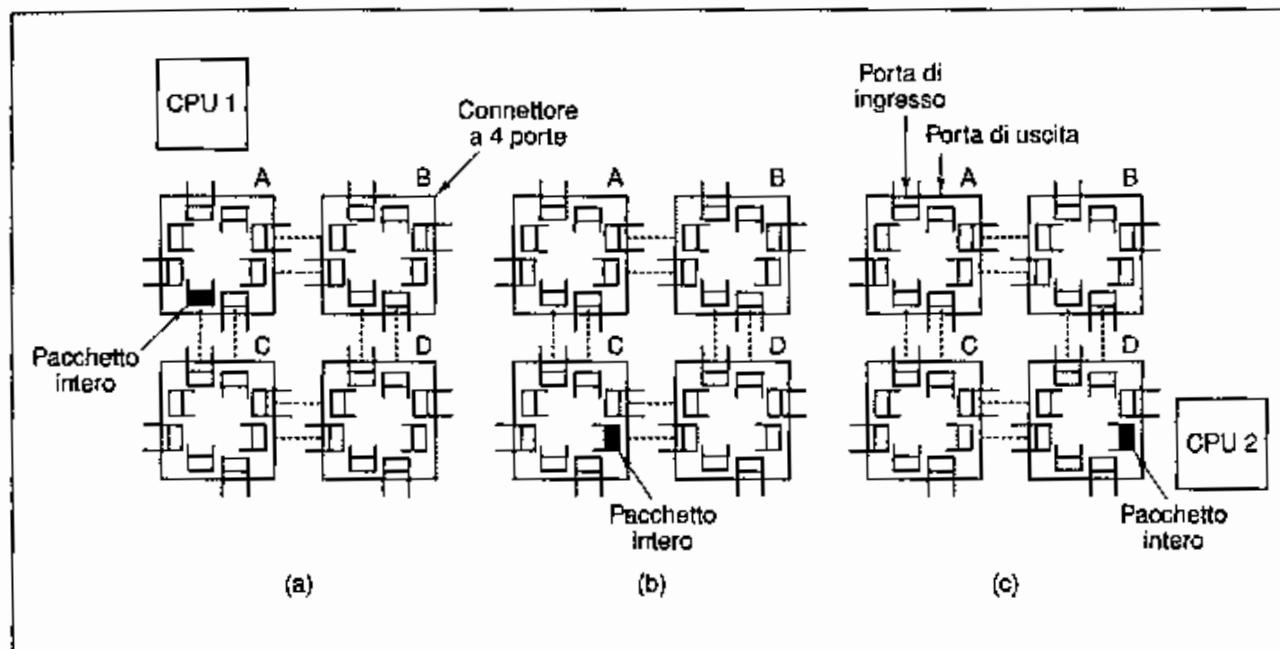


Figura 8.17 Comutazione di pacchetto store-and-forward.

parti di una determinata lunghezza massima, chiamata **pacchetto**. Lo schema di connessione (**commutazione di pacchetto store-and-forward**) è realizzato tramite pacchetti inviati al primo switch dalla scheda di rete del nodo sorgente, come vediamo nella Figura 8.17 (a). I bit arrivano uno ad uno e, quando l'intero pacchetto è arrivato, è copiato nello switch successivo lungo il suo cammino, come nella Figura 8.17 (b). Quando il pacchetto giunge allo switch del nodo di destinazione, come in Figura 8.17 (c), il pacchetto è copiato nell'interfaccia di rete di quel nodo, ed eventualmente nella sua RAM.

La commutazione di pacchetto store-and-forward è flessibile ed efficiente, però ha il problema di incrementare la latenza (ritardo) attraverso la rete di interconnessione. Si supponga che il tempo per spostare di un passo un pacchetto nella Figura 8.17 sia  $T$  ns. Poiché il pacchetto deve essere copiato quattro volte, perché passi dalla CPU 1 alla CPU 2 (attraverso A, C e D, fino alla CPU di destinazione), e non si può iniziare nessuna copia finché la precedente non è finita, la latenza attraverso la rete di interconnessione è  $4T$ . Una via d'uscita consiste nel progettare una rete ibrida con alcune delle proprietà della commutazione di circuito e alcune della commutazione di pacchetto. Ad esempio, ciascun pacchetto può essere diviso logicamente in unità più piccole, e non appena la prima unità arriva ad uno switch, può essere spedita a quello successivo, anche prima che la coda del pacchetto sia arrivata.

L'altro regime di commutazione, la **commutazione di circuito**, si realizza con un cammino prestabilito, attraverso tutti gli switch, dal primo a quello di destinazione; una volta stabilito il cammino, i bit sono spediti per tutto il tempo senza pausa dalla sorgente alla destinazione, e senza bufferizzazione intermedia negli switch. La commutazione di circuito comporta una fase di inizializzazione, che richiede un po' di tempo, ma è più veloce dopo che l'inizializzazione è stata completata; spedito il pacchetto, il cammino dev'essere cancellato. Una variante della commutazione di circuito, chiamato **wormhole routing**, suddivide ciascun pacchetto in sotto-pacchetti, e permette che il primo di questi inizi a fluire anche prima di stabilire l'intero cammino.

## Interfacce di rete

Tutti i nodi in un multicomputer hanno una scheda inserita per consentire la connessione del nodo alla rete di interconnessione. Il modo in cui queste schede sono costruite e connesse alla CPU principale e alla RAM, ha implicazioni notevoli per il sistema operativo, e ne esamineremo alcune. Questo materiale si basa in parte su Bhoedjang, 2000; altri riferimenti sono (Buzzard et al., 1996; Pakjin et al., 1997; Steenkiste, 1994; Von Eicken et al., 1992).

Praticamente in tutti i multicomputer, la scheda di interfaccia contiene della RAM per conservare i pacchetti in ingresso e in uscita, che, di solito, vanno copiati nella RAM della scheda di interfaccia, prima di poter essere trasmessi al primo switch. Questo meccanismo è motivato dal fatto che tutte le reti di interconnessione sono sincrone, cosicché, iniziata la trasmissione di un pacchetto, i bit devono fluire a velocità costante. Se il pacchetto è nella RAM principale, questo flusso continuo verso la rete non può essere garantito per l'altro traffico presente sul bus della memoria, e la RAM dedicata sulla scheda di interfaccia risolve questo problema. Questa architettura compare nella Figura 8.18.

Con i pacchetti d'ingresso si verifica lo stesso problema: i bit arrivano dalla rete ad una velocità costante e spesso assai elevata. Se la scheda di interfaccia non può memorizzarli in tempo reale quando arrivano, i dati andranno perduti, ed inoltre cercare di utilizzare il bus di sistema (come il bus PCI), per arrivare alla memoria RAM, è troppo rischioso. Poiché la scheda di rete di norma è inserita nel bus PCI, questa è la sola connessione della scheda alla RAM principale, e quindi entrerebbe in competizione per questo bus con il disco e ogni altro dispositivo di I/O. È più sicuro memorizzare i pacchetti in ingresso nella RAM privata della scheda di interfaccia, e copiarli più tardi nella RAM principale.

La scheda di interfaccia può avere uno o più canali DMA, o anche una CPU completa; i canali DMA possono copiare pacchetti fra la scheda di interfaccia e la RAM principale ad alta velocità, richiedendo trasferimenti di blocchi sul bus di sistema, e trasferendo quindi diverse parole senza dover richiedere il bus per ciascuna parola separatamente. Comunque, è proprio questo tipo di trasferimento di blocchi, che impegnà il bus di sistema per diversi cicli di bus, che rende necessaria la RAM sulla scheda di interfaccia.

Alcune schede di interfaccia possiedono una vera e propria CPU, eventualmente in aggiunta ad uno o più canali DMA. Questa architettura consente alla CPU principale di

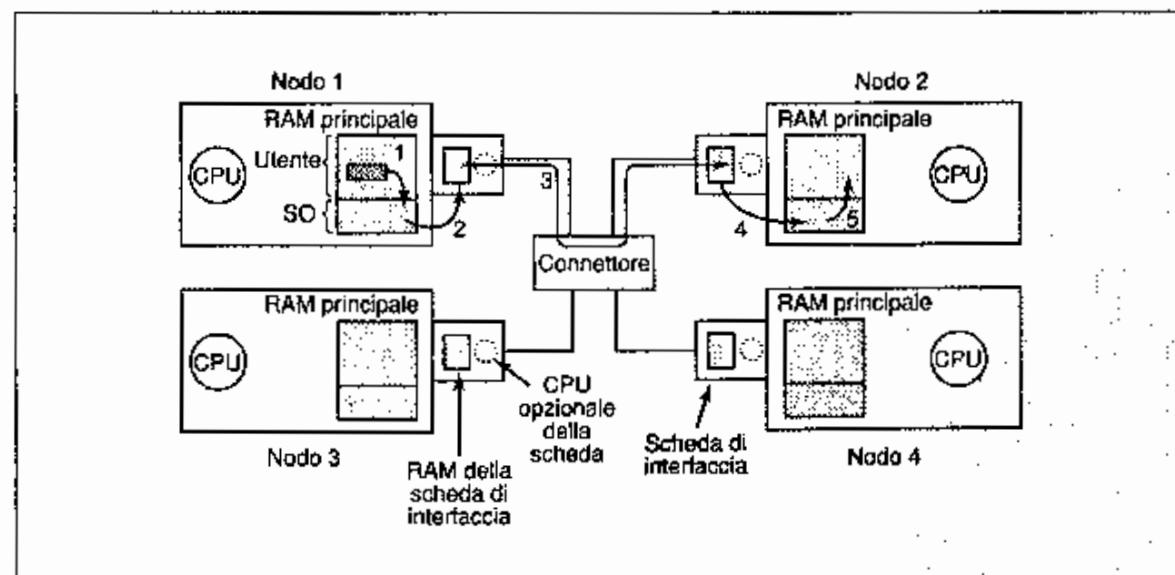


Figura 8.18 Posizione delle schede di interfaccia di rete in un multicomputer.

scaricare parte del lavoro alla scheda di rete, ad esempio la gestione della trasmissione affidabile (se l'hardware sottostante può perdere pacchetti), il multicasting (mandare un pacchetto a più di una destinazione), e la protezione in un sistema con processi multipli. In ogni caso, avere due CPU significa doverle sincronizzare, per evitare corse critiche, fatto che aggiunge overhead e significa più lavoro per il sistema operativo.

### 8.2.2 Software di comunicazione di basso livello

Il nemico della comunicazione ad alte prestazioni nei sistemi multicomputer è l'eccesso di copia dei pacchetti: nel migliore dei casi, avremo una copia dalla RAM alla scheda di interfaccia nel nodo sorgente, una copia dalla scheda di interfaccia del sorgente a quella della destinazione (se non avvengono memorizzazioni e inoltre lungo il cammino), e una copia da qui alla RAM di destinazione, per un totale di tre copie. In molti sistemi, tuttavia, la situazione è peggiore, in particolare, se la scheda di interfaccia è mappata nello spazio degli indirizzi virtuali del kernel e non in quello utente, un processo utente può mandare un pacchetto solo mediante una chiamata di sistema, che effettua una trap al kernel. I kernel possono dover copiare i pacchetti nella loro memoria, sia in entrata sia in uscita, ad esempio, per evitare fault di pagina mentre trasmettono sulla rete. Inoltre, probabilmente il kernel ricevente non sa dove mettere i pacchetti in arrivo, finché non ha la possibilità di esaminarli. Questi cinque passi di copia sono illustrati nella Figura 8.18.

Se le copie verso e dalla RAM dominano le prestazioni, le copie extra per e dal kernel possono duplicare il ritardo da una estremità all'altra, e dimezzare la larghezza di banda. Per evitare tale riduzione delle prestazioni, certi multicomputer mappano la scheda di interfaccia direttamente nello spazio utente, e permettono al processo utente di mettere i pacchetti direttamente sulla scheda, senza coinvolgere il kernel. Anche se l'approccio è vantaggioso per le prestazioni, comporta però due problemi.

Per prima cosa, che succede se diversi processi sono in esecuzione in un nodo e hanno bisogno di accedere alla rete per spedire pacchetti? Quale processo mette la scheda di interfaccia nel proprio spazio di indirizzi? Avere una chiamata di sistema per mappare una scheda dentro e fuori lo spazio degli indirizzi virtuali, è costoso, ma se solo un processo ottiene la scheda, come faranno gli altri a spedire i pacchetti? E cosa accade, se la scheda è mappata nello spazio degli indirizzi virtuali del processo A, e arriva un pacchetto per il processo B, specialmente se A e B hanno proprietari diversi, nessuno dei quali vuol fare uno sforzo per aiutare l'altro?

Una soluzione consiste nel mappare la scheda di interfaccia in tutti i processi che se ne servono, però occorre un meccanismo per evitare le corse critiche. Ad esempio, se A reclama un buffer sulla scheda di interfaccia, e poi, a causa dell'esaurimento del tempo di calcolo assegnatogli dallo scheduler, B passa in esecuzione e reclama lo stesso buffer, succede un disastro. Sarebbero necessari dei meccanismi di sincronizzazione, come i mutex, che però funzionano solo quando si suppone che i processi cooperino. In un ambiente time sharing, con diversi utenti tutti preoccupati che il proprio lavoro sia svolto, uno di loro potrebbe mettere il lock al mutex associato alla scheda, e non rilasciarlo mai. Si può dunque concludere che mappare la scheda di interfaccia nello spazio utente è realmente utile quando c'è un solo processo in esecuzione su ciascun nodo, a meno di prendere speciali precauzioni (ad esempio, mappare porzioni diverse della RAM dell'interfaccia nello spazio degli indirizzi dei diversi processi).

Il secondo problema è che il kernel può dover accedere esso stesso alla rete di interconnessione, ad esempio per accedere al file system di un nodo remoto. Avere il kernel che condivide la scheda di interfaccia con gli utenti, non è una buona idea, neanche utilizzando metodi di time sharing. Supponiamo che mentre la scheda è mappata nello spa-

zio utente, arrivi un pacchetto per il kernel, oppure che il processo utente mandi un pacchetto ad una macchina remota, fingendo di essere il kernel; l'architettura più semplice consiste nell'avere due interfacce di rete, una mappata nello spazio utente per il traffico delle applicazioni, l'altra nello spazio del kernel per il sistema operativo. Molti multicomputer fanno proprio questo.

## Comunicazione dell'interfaccia nodo-rete

Un altro problema è ricevere pacchetti sulla scheda di interfaccia: la via più rapida è servirsi del chip DMA per copiarli dalla RAM. Il problema, in questo caso, è che il DMA utilizza indirizzi fisici, piuttosto che virtuali, ed esegue indipendentemente dalla CPU. In primo luogo, benché un processo utente conosca certamente l'indirizzo virtuale di un pacchetto che vuol spedire, non conosce di solito quello fisico. Fare una chiamata di sistema per effettuare la mappatura da virtuale a fisico non è opportuno, poiché la ragione principale per mettere la scheda di interfaccia nello spazio utente consisteva proprio nell'evitare di fare una chiamata di sistema per ogni pacchetto da spedire.

Inoltre, se il sistema operativo decide di rimpiazzare una pagina mentre il chip DMA sta copiando un pacchetto da esse, saranno trasmessi dati sbagliati. Ancor peggio, se il sistema operativo rimpiazza una pagina mentre il chip DMA vi sta copiando un pacchetto, non solo il pacchetto in arrivo verrà perduto, ma si rovinerà anche una pagina di memoria innocente.

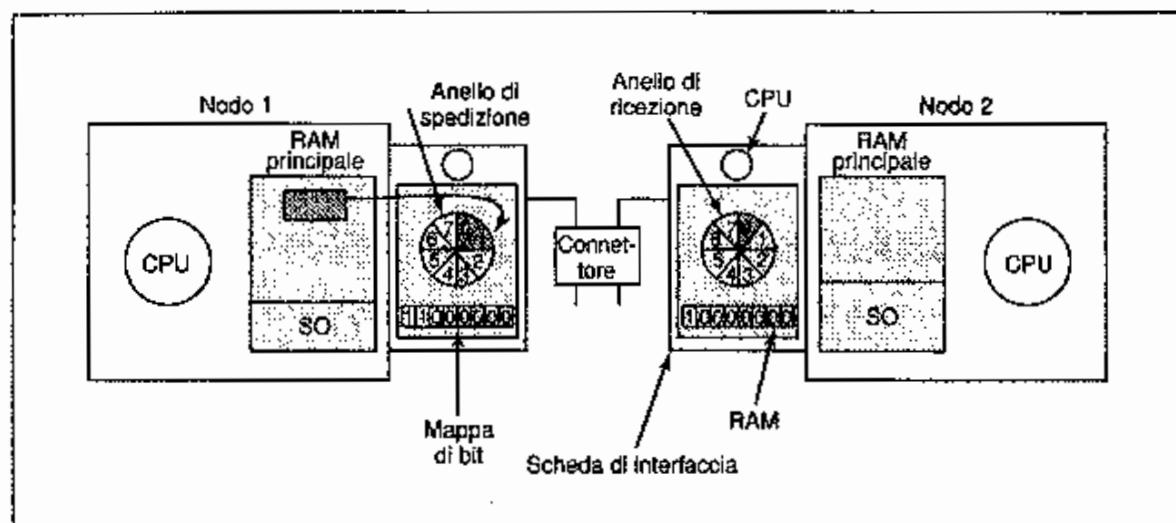
È possibile evitare questi problemi con chiamate di sistema per bloccare e sbloccare le pagine in memoria (pinning), marcandole come temporaneamente non paginabili. Comunque, effettuare una chiamata di sistema per bloccare la pagina contenente ciascun pacchetto uscente, e quindi dover fare una seconda chiamata più tardi per sbloccarla, è costoso; se i pacchetti sono piccoli, ad esempio 64 byte o meno, l'overhead per bloccare e sbloccare ogni buffer è proibitivo. Per grandi pacchetti, ad esempio 1 KB e oltre, può essere tollerabile, e per dimensioni intermedie, dipende dai dettagli dell'hardware (Bhoedjang 2000).

In teoria, lo stesso problema si verifica con il DMA da disco o altro dispositivo, ma poiché questi sono visti dal sistema operativo come buffer del kernel, è facile per il sistema evitare di paginare i buffer. Qui è l'utente che gestisce il DMA, e il sistema operativo ignora che la rimozione di una pagina potrebbe essere fatale, come invece gli è noto per l'I/O che esso stesso inizia. La ragione che rende accettabile utilizzare i buffer per l'I/O del disco, e non per la comunicazione multiprocessore, è che un ulteriore ritardo nell'ordine dei 20 µs è tollerabile per la latenza del disco, ma non per la latenza di comunicazione da processo a processo.

Il problema del DMA può essere evitato, facendo in modo che il processo utente blocchi prima una pagina all'avvio e richieda il suo indirizzo fisico. I pacchetti in uscita sono prima copiati lì, e quindi nell'interfaccia di rete. Questa copia in più, è però negativa, come la copia nel kernel.

Per questi motivi, utilizzare l'I/O programmato da e per la scheda di interfaccia è di solito la scelta più sicura, poiché ogni fault di pagina incontrato è un normale fault di pagina della CPU, ed il sistema operativo può gestirlo nel modo consueto. Quando avviene un fault di pagina, il loop di copia si ferma istantaneamente, e rimane bloccato finché il sistema operativo ha gestito il fault di pagina. Uno schema più sofisticato consiste nell'usare l'I/O programmato per piccoli pacchetti, e i DMA con sblocco e blocco delle pagine per quelli grandi.

Se le schede di interfaccia di rete hanno la loro propria CPU, (ad esempio le schede Myrinet), queste CPU si possono usare per accelerare la comunicazione. Occorre in ogni



**Figura 8.19** Utilizzo degli anelli di spedizione e ricezione per coordinare la CPU principale con la CPU della scheda.

modo porre attenzione ad evitare corse critiche fra la CPU principale e quella della scheda. Un modo per evitarle è illustrato nella Figura 8.19, dove il nodo 1 manda pacchetti e il nodo 2 li riceve, non necessariamente l'uno dall'altro. La struttura dati chiave per la sincronizzazione dei mittenti è l'anello di spedizione, per i riceventi è l'anello di ricezione; tutti i nodi li hanno entrambi, poiché tutti spediscono e ricevono, e ciascun anello ha spazio per  $n$  pacchetti. Per ogni anello, esiste una mappa di bit con  $n$  bit, possibilmente separati (vedi figura), oppure integrati negli anelli, al fine di indicare quali posizioni dell'anello siano correntemente valide.

Quando un mittente ha un nuovo pacchetto da spedire, controlla prima se esista una posizione disponibile nell'anello di spedizione; in caso negativo, deve aspettare, per evitare una sovrascrittura. Se c'è una posizione libera, copia il pacchetto nella successiva posizione disponibile, e alla fine imposta il bit corrispondente nella mappa di bit. Quando la CPU della scheda ha terminato il suo compito, controlla l'anello di spedizione: se contiene pacchetti, sceglie il più grosso e lo trasmette, e quando ha finito, azzera il bit corrispondente della mappa di bit. Poiché la CPU principale è la sola che imposta i bit, e la CPU sulla scheda è la sola che li azzera, non ci sono corse critiche. L'anello ricevente funziona in modo simmetrico, con la CPU della scheda che imposta un bit per indicare l'arrivo di un pacchetto, e la CPU principale che lo azzera per indicare che ha copiato il pacchetto e ha liberato il buffer.

Questo schema si può usare anche senza l'I/O programmato effettuato dalla CPU principale; in tal caso, l'entry dell'anello di spedizione non contiene il pacchetto, ma un puntatore al pacchetto nella RAM principale. Quando la CPU della scheda è pronta a trasmettere il pacchetto, lo preleva mettendolo sulla scheda di interfaccia, per mezzo dell'I/O programmato, oppure tramite DMA. In entrambi i casi, l'approccio funziona solo se la pagina contenente il pacchetto è bloccata.

### 8.2.3 Software di comunicazione a livello utente

I processi sulle diverse CPU di un multicomputer comunicano mandandosi dei messaggi; nella forma più semplice, tale scambio di messaggi è visibile ai processi utente. In altre parole, il sistema operativo fornisce un modo per mandare e ricevere messaggi, e le procedure di libreria rendono disponibili, ai processi utente, le chiamate di sistema sot-

tostanti. In una forma più sofisticata, l'effettivo scambio di messaggi è nascosto agli utenti, facendo apparire la comunicazione remota come una chiamata di procedura; studieremo entrambi questi metodi.

## Send e receive

I servizi di comunicazione possono essere ridotti al minimo a due chiamate (di libreria), una per mandare messaggi e l'altra per riceverli. La chiamata per mandare un messaggio potrebbe essere

```
send(dest, &mpt)
```

e la chiamata per ricevere il messaggio potrebbe essere

```
receive(addr, &mpt)
```

Il primo manda il messaggio puntato da *mpt* al processo identificato da *dest*, determinando il blocco del chiamante finché il messaggio non viene spedito; la seconda determina il blocco del chiamante finché non arriva un messaggio. Quando arriva, il messaggio è copiato nel buffer puntato da *mpt* e il chiamante è sbloccato; il parametro *addr* specifica l'indirizzo su cui il ricevente è in ascolto. Esistono molte varianti delle due procedure e dei loro parametri.

Come effettuare l'indirizzamento costituisce un problema: poiché i multicomputer sono statici, con numero delle CPU fissato, il modo più semplice per gestire l'indirizzamento è rendere *addr* un indirizzo in due parti, formato cioè dal numero di CPU e dal numero di processo o di porta sulla CPU indirizzata. Così ogni CPU può gestire i propri indirizzi senza altri conflitti potenziali.

## Primitive bloccanti e non bloccanti

Le primitive per lo scambio di messaggi che abbiamo descritto finora vengono dette primitive bloccanti (a volte anche primitive sincrone): quando un processo chiama la *send*, specifica la destinazione e un buffer da spedire a quella destinazione. Mentre il messaggio viene spedito il processo mittente si blocca (cioè viene sospeso); l'istruzione che segue la chiamata alla *send* non viene eseguita finché il messaggio non è stato completamente spedito, come in Figura 8.20 (a). In modo analogo una chiamata ad una *receive* non termina fino a quando il messaggio non è stato ricevuto e posto nel buffer puntato dal parametro; il processo rimane sospeso nella *receive* fino a quando arriva un messaggio, anche nel caso in cui occorrono delle ore. In alcuni casi, il ricevente può specificare da chi desidera ricevere, nel qual caso rimane bloccato fino a quando non arrivi un messaggio da quel mittente.

Un'alternativa alle primitive bloccanti è rappresentata dalle primitive non bloccanti (a volte chiamate anche primitive asincrone). La *send* non bloccante restituisce immediatamente il controllo al chiamante, prima che il messaggio venga spedito; il vantaggio di questo schema deriva dal fatto che il mittente può continuare il calcolo in parallelo alla trasmissione del messaggio, anziché lasciare disoccupata la CPU (nell'ipotesi che nessun altro processo possa andare in esecuzione). La scelta fra primitive bloccanti e non bloccanti (cioè se debbano essere disponibili le une o le altre) viene fatta normalmente dai progettisti del sistema, sebbene in alcuni (pochi) sistemi esistano entrambe le possibilità e l'utente possa scegliere quella che più gli aggrada.

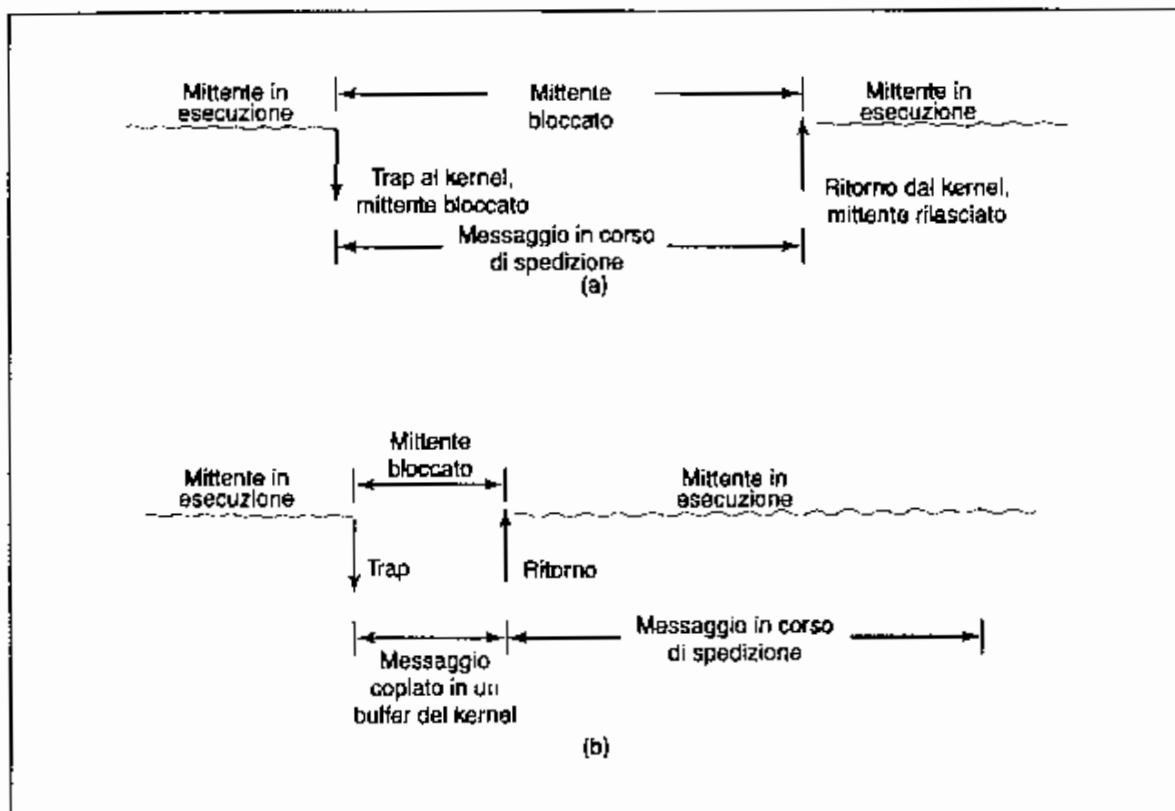


Figura 8.20 [a] Una primitiva send bloccante. [b] Una primitiva send non bloccante.

In ogni caso, il vantaggio in termini di prestazioni offerto dalle primitive non bloccanti viene controbilanciato da uno svantaggio piuttosto serio: il mittente non può modificare il buffer che contiene il messaggio fino a quando il messaggio non è stato spedito; le conseguenze che derivano dal fatto che un processo sovrascriva un messaggio durante la trasmissione sono troppo orribili per prenderle in considerazione, e peggio ancora, il processo mittente non ha nessuna idea di quando venga completata la trasmissione, quindi non può mai sapere quando possa riusare il buffer senza fare danni; d'altra parte è veramente difficile evitare per sempre di usare un certo buffer.

Ci sono tre vie d'uscita possibili. La prima è quella in cui il kernel si crea una copia del messaggio in un suo buffer interno, per poi permettere al processo di continuare nell'esecuzione, come in Figura 8.20 (b). Dal punto di vista del mittente, questo schema è lo stesso della chiamata bloccante: non appena ottiene di nuovo il controllo, è libero di usare di nuovo il buffer; naturalmente il messaggio non sarà ancora stato spedito, ma il processo mittente non risulta per questo in alcun modo rallentato. Lo svantaggio di questo metodo sta nel fatto che ogni messaggio in uscita dev'essere copiato dallo spazio utente allo spazio del kernel; in molte interfacce di rete, esso dev'essere copiato più tardi nel buffer dell'hardware di trasmissione, così la prima copia è sostanzialmente inutile, e la seconda degrada in modo sensibile le prestazioni del sistema.

La seconda soluzione consiste nel generare un'interruzione verso il mittente quando il messaggio sia stato effettivamente spedito per informarlo che il buffer è nuovamente disponibile. In questo caso, non è necessaria nessuna copia, il che comporta un risparmio di tempo, ma la gestione delle interruzioni a livello utente rende la programmazione difficile, tortuosa e soggetta a corse critiche, rendendole irriproducibili, e quasi impossibile farne il debug.

La terza soluzione consiste nel marcare il buffer come copia in scrittura (copy-on-write), cioè marcare il buffer in sola lettura finché il messaggio non è stato spedito; se il buffer viene riutilizzato prima che il messaggio sia spedito viene effettuata una copia. Il problema di questa soluzione è che, salvo che il buffer non sia isolato nella propria pagina, scritture in variabili vicine al buffer stesso forzeranno una copia. Sono inoltre necessarie operazioni di gestione aggiuntive perché la spedizione di un messaggio ha implicitamente effetto sullo stato di lettura/scrittura di una pagina, ed infine, prima o poi la pagina verrà nuovamente riscritta, causando una copia che può non essere più necessaria.

Quindi, dal punto di vista del mittente le scelte sono

1. *send* bloccante (CPU inattiva durante la trasmissione del messaggio).
2. *send* non bloccante con copia (spreco di tempo di CPU per la copia).
3. *send* non bloccante con interruzione (rende difficile la programmazione).
4. Copia in scrittura (con probabile copia aggiuntiva).

In condizioni normali la prima scelta è la migliore, specialmente se sono disponibili thread multipli, nel qual caso, mentre un thread è bloccato su una *send*, gli altri possono continuare a lavorare; non si richiede neppure un buffer del kernel. Inoltre, come si può vedere confrontando le Figura 8.20 (a) e (b), il messaggio sarà spedito più rapidamente se non è richiesta alcuna copia.

Vogliamo però sottolineare che altri autori distinguono le primitive sincrone e asincrone sulla base di criteri diversi. Secondo questo diverso punto di vista, una primitiva è sincrona solo se il mittente è bloccato fino al ricevimento del messaggio, e a conferma di avvenuta ricezione rispedita al mittente (Andrews 1991). Nell'ambito della comunicazione in tempo reale "sincrono" ha ancora un altro significato, il che può generare una certa confusione.

Così come la *send*, anche la *receive* può essere bloccante o non bloccante. Una primitiva bloccante sospende il chiamante finché il messaggio non è arrivato, e se sono disponibili thread multipli, questo è un approccio semplice. In alternativa, una *receive* non bloccante dice al kernel dove si trova il buffer e ritorna quasi immediatamente. Si può utilizzare un'interruzione per segnalare che un messaggio è arrivato, ma le interruzioni sono difficili da programmare e anche piuttosto lente, quindi è preferibile che il ricevente effettui il poll per i messaggi in ingresso, utilizzando una procedura, *poll*, che rileva se ci siano dei messaggi in attesa. In caso affermativo, il chiamante può chiamare *get\_message*, che restituisce il primo messaggio arrivato. In alcuni sistemi, il compilatore può inserire delle chiamate di *poll* nel codice nei posti appropriati, benché conoscere quanto spesso effettuare il poll sia difficile.

Un'ulteriore possibilità è uno schema in cui l'arrivo di un messaggio genera la creazione spontanea di un thread nello spazio degli indirizzi del processo ricevente. Tale thread è chiamato **thread di pop-up**: esso esegue una procedura precedentemente specificata, il cui parametro è un puntatore al messaggio in arrivo. Dopo aver gestito il messaggio, termina e viene automaticamente distrutto.

Una variante di quest'idea consiste nell'eseguire il codice di ricezione direttamente nel gestore dell'interruzione, senza prendersi il grattacapo di creare un thread di pop-up. Per rendere questo schema ancora più veloce, il messaggio stesso contiene l'indirizzo del gestore dell'interruzione, cosicché, quando arriva un messaggio, il gestore stesso può essere chiamato con poche istruzioni. In questo caso, il maggior vantaggio è che non è necessaria alcuna copia: il gestore preleva il messaggio dalla scheda di interfaccia e lo elabora al volo. Tale schema è noto come **schema a messaggi attivi** (Von

Eicken et al., 1992); poiché ciascun messaggio contiene l'indirizzo del gestore dell'interruzione, i messaggi attivi funzionano solo quando mittenti e riceventi si fidano completamente l'uno dell'altro.

### 8.2.4 Chiamate di procedura remota

Benché il modello di scambio di messaggi fornisca un modo conveniente per strutturare un sistema operativo per multicomputer, soffre di un difetto incurabile: il paradigma base su cui è costruito l'intero processo di comunicazione è l'input/output. Le procedure *send* e *receive* sono fondamentalmente impegnate nell'effettuare I/O, e molti credono che l'I/O sia il modello di programmazione sbagliato.

Il problema è noto da tempo, ma poco è stato fatto finché un articolo di Birrell e Nelson (1984) ha introdotto un modo nuovo di affrontarlo. Benché l'idea sia veramente semplice (una volta che qualcuno l'abbia pensata), le implicazioni spesso appaiono sottili: in questa sezione esamineremo il concetto, la sua implementazione, i suoi punti forti e quelli deboli.

In poche parole, Birrell e Nelson hanno suggerito di permettere ai programmi di chiamare procedure su altre CPU. Quando un processo sulla macchina 1 chiama una procedura sulla macchina 2, il processo chiamante su 1 è sospeso, e l'esecuzione della procedura chiamata avviene su 2. L'informazione può essere trasportata dal chiamante al chiamato attraverso i parametri, e può tornare indietro attraverso i risultati della procedura; nessuno scambio di messaggi o I/O è visibile al programmatore. Questa tecnica è nota come **RPC** (Remote Procedure Call, chiamata di procedura remota) ed è alla base di una grande quantità di software per multicomputer. Tradizionalmente, la procedura chiamante è nota come **client**, quella chiamata come **server**, e anche noi useremo questa terminologia.

L'idea alla base di RPC è di far sembrare una chiamata di procedura remota il più possibile come una chiamata locale. Nella forma più semplice, per chiamare una procedura remota il programma client deve essere collegato ad una piccola procedura di libreria, chiamata **client stub**, che rappresenta la procedura server nello spazio degli indirizzi del client, poi allo stesso modo, il server è collegato con una procedura chiamata **server stub**; queste procedure nascondono il fatto che la chiamata di procedura dal client al server non è locale.

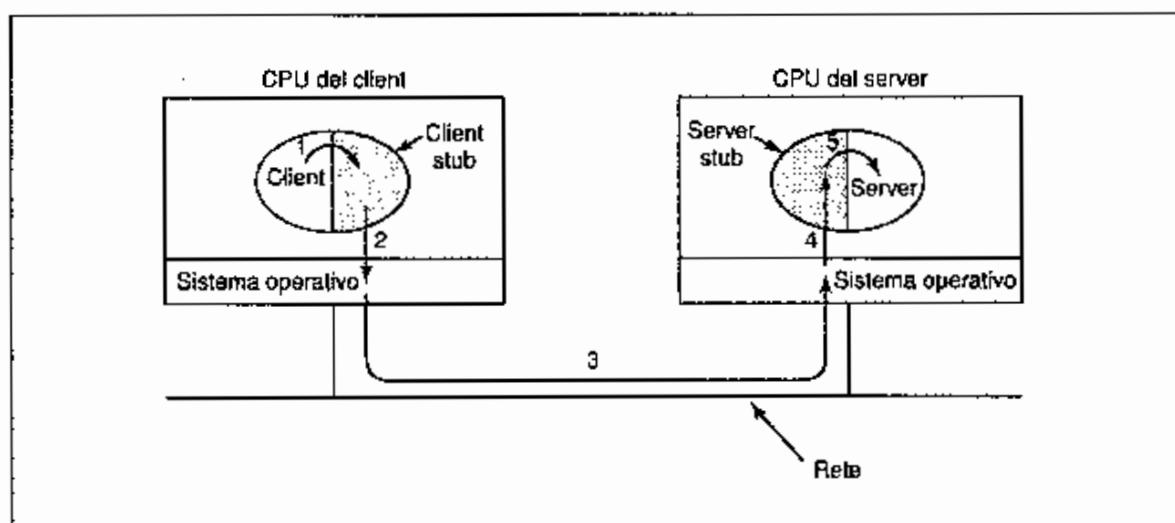


Figura 8.21 Passi di una chiamata di procedura remota. Gli stub sono ombreggiati in grigio.

I passi effettivi per fare una RPC compaiono nella Figura 8.21: il passo 1 è la chiamata da parte del client del client stub, che è una chiamata di procedura locale, con i parametri messi sullo stack nel modo consueto. Il passo 2 consiste nell'impacchettamento dei parametri in un messaggio e nell'effettuare una chiamata di sistema per spedire il messaggio; impacchettare i parametri è detto **marshaling**. Nel passo 3, il kernel spedisce il messaggio dalla macchina client a quella server, e nel passo 4 il kernel passa il pacchetto in arrivo al server stub (che in precedenza avrebbe normalmente chiamato *receive*). Infine, nel passo 5 il server stub chiama la procedura del server. La risposta effettua lo stesso cammino in direzione opposta.

L'elemento chiave da osservare, è che la procedura client, scritta dall'utente, non effettua altro che una chiamata di procedura normale (cioè locale) al client stub, che ha lo stesso nome della procedura del server; poiché la procedura del client e il client stub sono nello stesso spazio degli indirizzi, i parametri sono passati nel modo usuale. Allo stesso modo, la procedura del server è chiamata da una procedura nel suo spazio degli indirizzi, con i parametri che si aspetta; per la procedura del server, non c'è nulla di inusuale. Così, invece di effettuare I/O tramite *send* e *receive*, la comunicazione remota avviene per mezzo di una normale chiamata di procedura.

## Problemi implementativi

Nonostante l'eleganza concettuale di RPC, esistono diversi problemi nascosti. Un primo problema significativo è quello dei parametri puntatore: normalmente non è un problema passare un puntatore ad una procedura, perché la procedura chiamata può usare il puntatore nello stesso modo di quella chiamante, in quanto le due procedure risiedono nello stesso spazio degli indirizzi virtuali. Con RPC il passaggio dei puntatori è impossibile, perché il client ed il server hanno un diverso spazio degli indirizzi.

In alcuni casi si possono usare degli espedienti per rendere possibile il passaggio di puntatori: si supponga che il primo parametro sia un puntatore ad un intero *k*. La client stub può effettuare il marshaling di *k* e spedirlo al server; il server stub quindi crea un puntatore a *k* e lo passa alla procedura del server, proprio come si aspetta. Quando la procedura del server ritorna il controllo al server stub, quest'ultima rispedisce *k* indietro al client, dove il nuovo valore di *k* è copiato su quello vecchio, nel caso il server lo avesse cambiato. In effetti la sequenza di chiamata standard con passaggio dei parametri per riferimento (call-by-reference) è stata rimpiazzata da una chiamata copia-ricopia (copy-restore). Disgraziatamente, questo espediente non sempre funziona, per esempio se il puntatore punta ad un grafo o ad altre strutture dati complesse; per questa ragione, si devono porre alcune restrizioni sul passaggio di parametri per procedura chiamate in remoto.

Il secondo problema capita con i linguaggi debolmente tipati, come il C, in cui è perfettamente legale scrivere una procedura che calcola il prodotto scalare di due vettori (array) senza specificare la lunghezza, ad esempio decidendo che esiste un valore speciale, noto solo a chiamante e chiamato, che indica la fine del vettore. In questo caso, è sostanzialmente impossibile che un client stub possa fare il marshaling dei parametri, se non ne conosce la dimensione.

In terzo luogo, non sempre è possibile dedurre i tipi dei parametri, nemmeno con una specifica formale, o dal codice stesso. Un esempio è *printf*, che può avere qualsiasi numero di parametri (almeno uno), che possono essere una miscela arbitraria di interi, short, long, caratteri, stringhe, numeri in virgola mobile di varia lunghezza, e altro. Cercare di chiamare una *printf* come una procedura remota, sarebbe praticamente impossibile a causa della permissività del C. In ogni caso, una regola che affermi che RPC possa essere usa-

to a condizione che non si programmi in C (o in C++), non sarebbe popolare.

Un quarto problema è relativo all'uso delle variabili globali. Di norma la procedura chiamante e quella chiamata possono comunicare tramite le variabili globali, oltre che per mezzo dei parametri. Se la procedura chiamata viene spostata in una macchina remota, il codice fallirà, perché le variabili globali non sono più condivise.

Tutto ciò non significa che RPC sia senza speranza, infatti, è ampiamente usata, ma sono necessarie alcune restrizioni e molta attenzione per fare in modo che lavori bene in pratica.

### 8.2.5 Memoria condivisa distribuita

Benché RPC sia interessante, molti programmatore preferiscono ancora un modello di memoria condivisa, e lo vorrebbero utilizzare anche su un multicomputer. Singolarmente, è possibile preservare l'illusione della memoria condivisa anche quando non esiste in realtà, utilizzando una tecnica chiamata DSM (Distributed Shared Memory, memoria condivisa distribuita). (Li, 1986; Li e Hudak, 1989). Con DSM ciascuna pagina si trova in una delle memorie della Figura 8.1, ogni macchina ha la propria memoria virtuale, e le proprie tabelle delle pagine. Quando una CPU effettua una LOAD o una STORE su una pagina che non ha, avviene una trap al sistema operativo, che, quindi, localizza la pagina, e chiede alla CPU che la possiede correntemente di invalidare la pagina, e spedirla

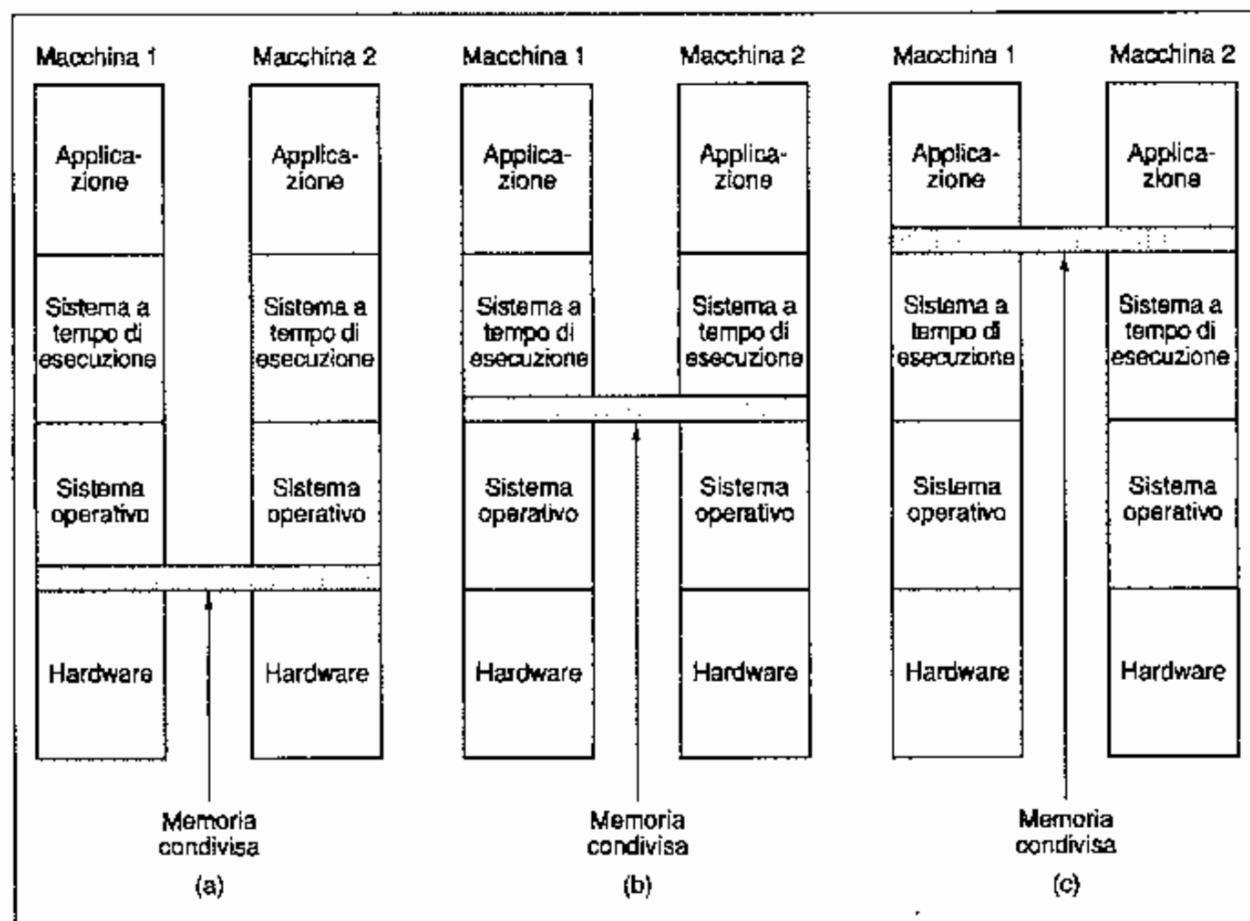


Figura 8.22 Diversi livelli in cui la memoria condivisa può essere implementata. (a) Hardware. (b) Il sistema operativo, (c) Software a livello utente.

sulla rete di interconnessione. Quando arriva, la pagina viene mappata e viene fatta ripartire l'istruzione che aveva provocato il fault; in effetti, il sistema operativo sta soddisfacendo i fault di pagina dalla RAM remota, invece che dal disco locale, e dal punto di vista dell'utente, sembra che la macchina abbia una memoria condivisa.

La differenza fra una vera memoria condivisa e DSM è illustrata nella Figura 8.22. Nella Figura 8.22 (a), si vede un vero multiprocessore con memoria fisica condivisa implementata tramite hardware; nella Figura 8.22 (b) si vede DSM implementata dal sistema operativo, e nella Figura 8.22 (c) troviamo ancora un'altra forma di memoria condivisa, implementata da livelli superiori del software. Rituneremo in seguito su questo terzo tipo, ma per ora esamineremo DSM.

Guardiamo più in dettaglio come funziona DSM: in un sistema DSM lo spazio degli indirizzi è suddiviso in pagine, distribuite in tutti i nodi del sistema, e quando una CPU si riferisce ad un indirizzo che non è locale, avviene una trap, il software DSM preleva la pagina che contiene l'indirizzo, e fa ripartire l'istruzione che ha provocato il fault, che ora si completerà con successo. Questo concetto è illustrato nella Figura 8.23 (a), per uno spazio degli indirizzi con sedici pagine e quattro nodi, ciascuno in grado di mantenere quattro pagine.

In questo esempio, se la CPU 0 si riferisce ad istruzioni (o dati) nelle pagine 2, 5 o 9, i riferimenti sono fatti localmente, ma i riferimenti ad altre pagine causano delle trap. Ad esempio, un riferimento ad un indirizzo nella pagina 10, causerà una trap al software DSM, che quindi muoverà la pagina 10 dal nodo 1 al nodo 0, come vediamo nella Figura 8.23 (b).

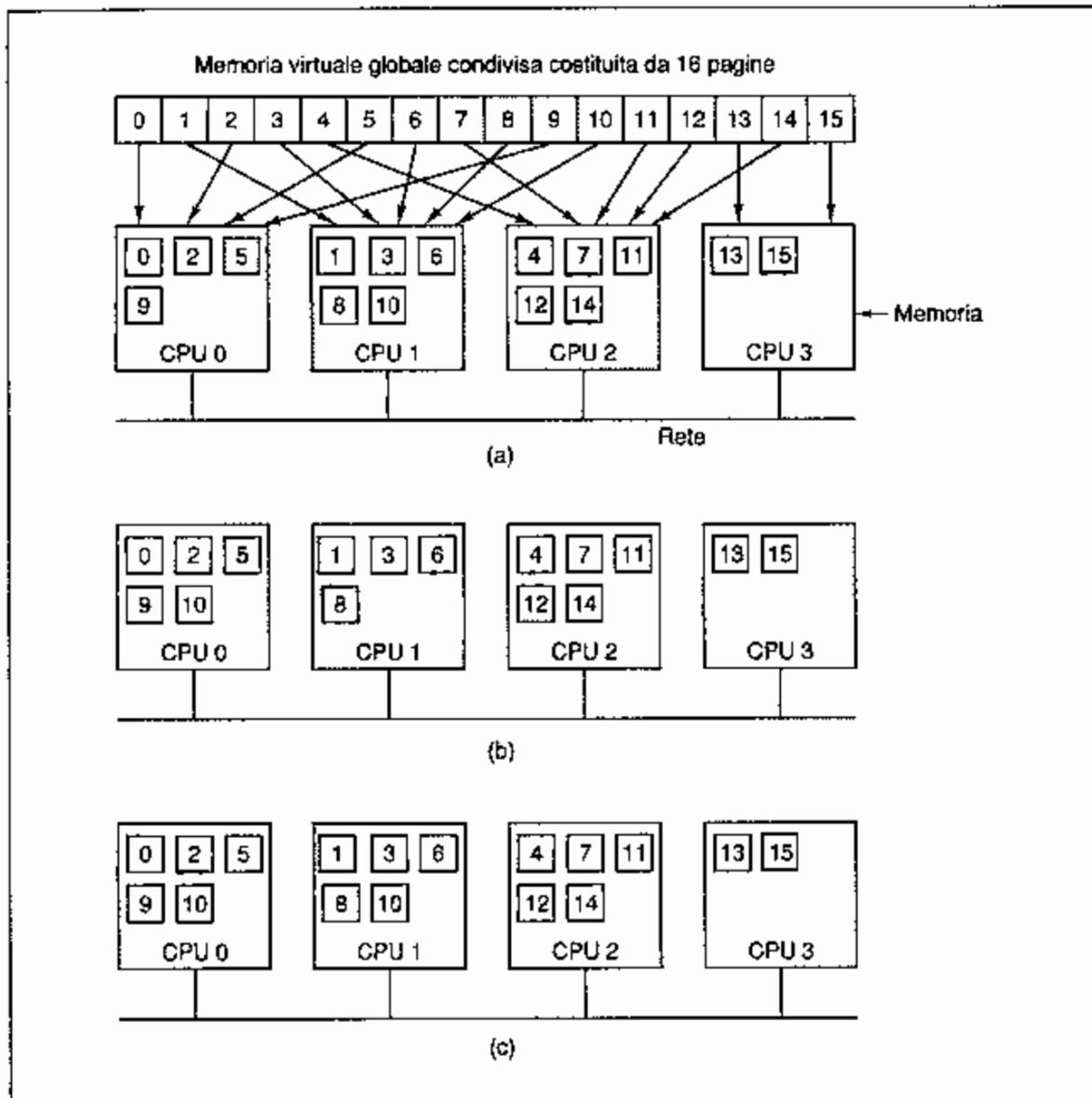
## Replicazione

Un miglioramento del sistema base che eleva anche di molto le sue prestazioni, consiste nel replicare le pagine in sola lettura, ad esempio testo del programma, costanti a sola lettura o altre strutture dati a sola lettura. Ad esempio, se la pagina 10 della Figura 8.23 è una sezione di testo di programma, il suo uso tramite la CPU 0 può far spedire una copia alla CPU 0, senza che l'originale nella memoria della CPU 1 sia disturbato, come si vede nella Figura 8.23 (c). In tal modo le CPU 0 e 1 possono far riferimento entrambe alla pagina 10 ogni volta che sia necessario, senza che avvengano trap di fault di pagina.

Una seconda possibilità è la replicazione non soltanto delle pagine in sola lettura, ma anche di tutte le altre. Fino a quando si effettuano solo letture, non esiste alcuna vera differenza fra la replicazione di una pagina in sola lettura e quella di una pagina in lettura-scrittura, e se una pagina replicata è improvvisamente modificata, occorre intervenire per evitare di avere diverse copie, non consistenti. Discuteremo nelle sezioni seguenti le modalità per ovviare all'inconsistenza.

## Condivisione falsa

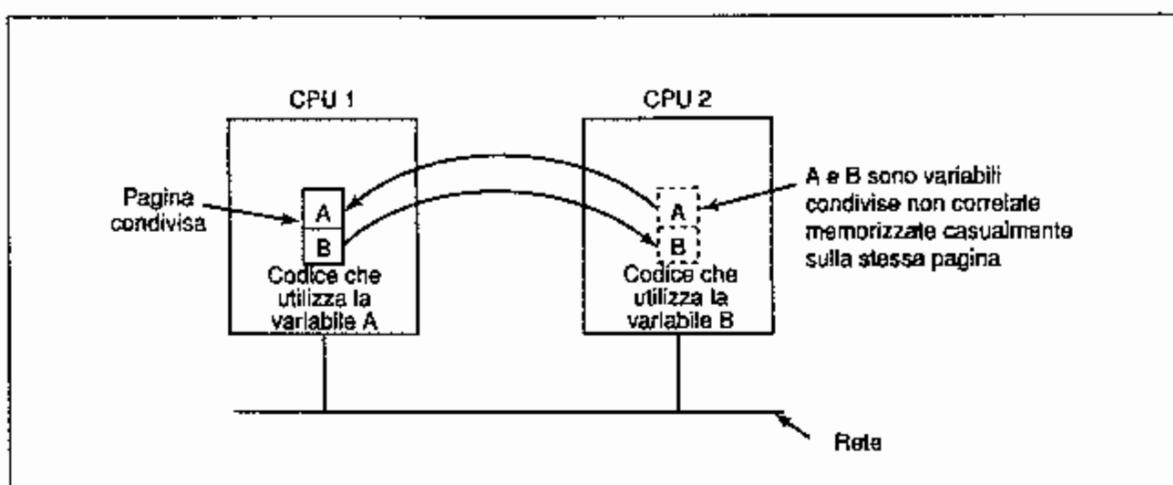
Per certe caratteristiche chiave, i sistemi DSM sono simili ai multiprocessori: in entrambi i sistemi, quando una parola di memoria non locale è riferita, una parte di memoria contenente la parola è prelevata dalla sua posizione corrente, e posta sulla macchina che ha fatto il riferimento (rispettivamente in memoria principale o cache). Un problema di progettazione importante è stabilire quanto debba essere grande la porzione di memoria: nei multiprocessori, la dimensione del blocco della cache è di solito 32 o 64 byte, per evitare di impegnare il bus con trasferimenti troppo lunghi, ma nei sistemi DSM l'unità dev'essere un multiplo della dimensione della pagina (perché la MMU funziona con le pagine), ma può essere 1, 2, 4 o più pagine. In effetti, ciò simula una dimensione di pagina maggiore.



**Figura 8.23** (a) Pagine dello spazio degli indirizzi distribuiti fra quattro macchine. (b) Situazione dopo che la CPU 1 ha fatto riferimento alla pagina 10. (c) Situazione se la pagina 10 è in sola lettura, e si utilizza la replicazione.

Ci sono svantaggi e vantaggi per una dimensione maggiore di pagina di DSM; il vantaggio maggiore è che, poiché il tempo di inizializzazione per un trasferimento di rete è abbastanza rilevante, non ci vuole veramente molto più tempo per trasferire 4096 byte piuttosto che 1024. Trasferendo i dati in grandi unità, è spesso possibile ridurre il numero dei trasferimenti, quando si deve spostare una larga porzione dello spazio degli indirizzi, e questa proprietà è particolarmente importante perché molti programmi esibiscono località di riferimento, ovvero, se un programma ha fatto riferimento ad una parola in una pagina, è probabile che farà riferimento ad altre parole nella stessa pagina nell'immediato futuro.

D'altro canto, la rete sarà più a lungo impegnata con un trasferimento maggiore, bloccando altri fault causati da altri processi. Inoltre una dimensione di pagina effettiva troppo grande, introduce un nuovo problema, chiamato **condivisione falsa**, illustrato nella



**Figura 8.24** Condivisione falsa di una pagina contenente due variabili non correlate.

Figura 8.24. Abbiamo una pagina che contiene due variabili condivise non correlate, *A* e *B*; il processore 1 fa un utilizzo pesante di *A*, leggendo e scrivendo su di essa. Allo stesso modo, il processore 2 utilizza spesso *B*: in questa situazione la pagina che contiene entrambe le variabili, viaggerà di continuo fra le due macchine.

Benché le variabili non siano correlate, appaiono per caso nella stessa pagina, così, quando un processo ne utilizza una, ottiene anche l'altra. Più grande è la dimensione effettiva della pagina, più spesso avremo condivisione falsa, e, al contrario, più piccola sarà, meno spesso si verificherà la condivisione falsa. Negli ordinari sistemi di memoria virtuale non è presente nulla di analogo.

Compilatori intelligenti, che comprendono il problema e mettono le variabili nello spazio degli indirizzi in modo opportuno, possono aiutare a ridurre la condivisione falsa e ad aumentare le prestazioni, ma è più facile a dirsi che a farsi. Inoltre, se la condivisione falsa consiste nel fatto che il nodo 1 utilizza un elemento di un array, ed il nodo 2 ne utilizza un altro dello stesso array, anche un compilatore intelligente può fare poco per eliminare il problema.

### Ottenere consistenza sequenziale

Se le pagine scrivibili non sono replicate, ottenere la consistenza non è un problema: esiste esattamente una copia di ciascuna pagina scrivibile, ed è spostata avanti ed indietro dinamicamente se necessario. Poiché non è sempre possibile stabilire in anticipo quali pagine siano scrivibili, in molti sistemi DMS, quando un processo cerca di leggere una pagina remota, viene fatta una copia locale e sia la copia remota sia quella locale sono inizializzate nelle rispettive MMU come in sola lettura. Fino a quando tutti i riferimenti sono letture, tutto funziona bene.

Comunque, se un processo tenta di scrivere su una pagina replicata, sorge un problema potenziale di consistenza, perché cambiare una copia e lasciare le altre inalterate è inaccettabile. Questa situazione è analoga a quella di un multiprocessore, quando una CPU tenta di modificare una parola presente in diverse cache. In quest'ultimo caso la soluzione è che la CPU che sta per scrivere metta prima un segnale sul bus, dicendo a tutte le altre CPU di scaricare la loro copia dalla cache. I sistemi DSM funzionano appunto nello stesso modo: prima che una pagina condivisa sia scritta, si invia un messaggio a tutte le altre CPU che hanno una copia della pagina, dicendo loro di invali-

darla e scaricarla. Dopo che tutte hanno risposto positivamente, la CPU originale può effettuare la scrittura.

È anche possibile tollerare copie multiple di pagine scrivibili in situazioni attamente circoscritte; ad esempio permettendo ad un processo di acquisire un lock su una porzione dello spazio degli indirizzi virtuale, e quindi di effettuare diverse operazioni di lettura e scrittura sulla memoria bloccata. Nel momento in cui il lock è rilasciato, i cambiamenti sono propagati alle altre copie; e se solo una CPU può bloccare una pagina in un determinato momento, questo schema preserva la consistenza.

In alternativa, quando una pagina potenzialmente scrivibile è scritta davvero per la prima volta, viene fatta una copia pulita, salvata sulla CPU che sta facendo la scrittura; quindi si possono acquisire i lock sulla pagina, aggiornarla, e rilasciare i lock. Più tardi, quando un processo su una macchina remota cerca di acquisire il lock della pagina, la CPU che ha scritto in precedenza sulla pagina stessa, confronta lo stato corrente della pagina con la copia pulita che aveva salvato, e costruisce un messaggio che elenca tutte le parole che ha cambiato. La lista è inviata alla CPU remota, in modo che essa possa aggiornare la propria copia, invece di invalidarla (Keleher et al., 1994).

## 8.2.6 Schedulazione dei multicomputer

In un multiprocessore tutti i processi risiedono nella stessa memoria. Quando una CPU finisce il suo task corrente, seleziona un processo e lo esegue: in linea di principio, tutti i processi sono potenziali candidati. Su un multicomputer la situazione è piuttosto diversa: ciascun nodo, infatti, ha memoria e insieme di processi propri, e la CPU 1 non può improvvisamente decidere di eseguire un processo che si trova sul nodo 4 senza prima fare una certa quantità di lavoro per acquisirlo. Questa differenza significa che la schedulazione sui multicomputer è più facile, ma l'allocazione dei processi ai nodi è più importante: studieremo questi problemi.

La schedulazione dei multicomputer è in un certo senso simile a quella dei multiprocessori, ma non tutti gli algoritmi dei primi si applicano ai secondi. L'algoritmo più semplice usato per i multiprocessori, mantenere una sola lista centralizzata dei processi pronti, in ogni caso non funziona, poiché ciascun processo si può eseguire solo sulla CPU dove si trova correntemente, e quando si crea un nuovo processo, è possibile scegliere dove metterlo, ad esempio per bilanciare il carico.

Poiché ciascun nodo ha i suoi processi, si può usare qualsiasi algoritmo di schedulazione locale. È anche possibile, tuttavia, usare la schedulazione gang, come la si utilizza su un multiprocessore, perché richiede soltanto un accordo iniziale su quali processi eseguire nei diversi intervalli di tempo, e una coordinazione dell'inizio degli intervalli di tempo.

## 8.2.7 Bilanciamento del carico

Non ci sono molte osservazioni da fare sulla schedulazione dei multicomputer: una volta che un processo è stato assegnato ad un nodo, entra in azione un algoritmo di schedulazione locale, a meno che non si stia usando la schedulazione gang. Proprio perché esistono poche possibilità di controllare un processo assegnato ad un nodo, è importante decidere quale processo debba essere eseguito su un determinato nodo; nei sistemi multiprocessore, invece, tutti i processi vivono nella stessa memoria, ed è possibile sederli su qualsiasi CPU. Di conseguenza, è opportuno studiare come sia concretamente possibile assegnare i processi ai nodi: gli algoritmi e l'euristica per effettuare tale assegnazione, sono noti come algoritmi di allocazione del processore e, nel corso degli anni,

ne sono stati proposti molti, che differiscono in ciò che ipotizzano come noto e negli obiettivi.

Le proprietà di un processo che potrebbero essere note comprendono: il fabbisogno di CPU, l'utilizzo di memoria e la quantità di comunicazione con ogni altro processo; i possibili obiettivi sono: la minimizzazione dei cicli di CPU sprecati per la carenza di lavoro locale, la minimizzazione della larghezza di banda di comunicazione totale, e condizioni eque per gli utenti e i processi. Prenderemo quindi in esame alcuni algoritmi, per avere un'idea di ciò che è possibile fare.

### Un algoritmo deterministico basato sulla teoria dei grafi

Una classe di algoritmi ampiamente studiata riguarda sistemi costituiti da processi con fabbisogni di CPU e memoria noti, e una matrice nota che memorizza le quantità di traffico fra ciascuna coppia di processi. Se il numero di processi è maggiore del numero di CPU,  $k$ , diversi processi dovranno essere assegnati alla stessa CPU. L'idea è di effettuare quest'assegnazione in modo da minimizzare il traffico della rete.

Il sistema si può rappresentare come un grafo pesato, con un processo associato a ciascun vertice, e ciascun arco che rappresenta il flusso dei messaggi fra due processi. Matematicamente, il problema si riduce quindi a trovare un modo per partizionare (cioè tagliare) il grafo in  $k$  sottografi disgiunti, soggetti a determinati vincoli (ad esempio, fabbisogno di CPU e memoria entro determinati limiti per ciascun sottografo). Per ciascuna soluzione che soddisfi i vincoli, gli archi compresi interamente entro un singolo sottografo rappresentano la comunicazione all'interno di una macchina, e possono essere ignorati; gli archi che vanno da un sottografo ad uno altro, rappresentano il traffico di rete, e l'obiettivo è quindi di trovare la partizione che minimizza il traffico di rete, soddisfacendo tutti i vincoli. Ad esempio, la Figura 8.25 mostra un sistema di nove processi, da A ad I, con ciascun arco etichettato con il carico medio di comunicazione fra due processi (ad esempio, in Mbps).

Nella Figura 8.25 (a) si è partizionato il grafo con i processi A, E e G sul nodo 1, i processi B, F e H sul nodo 2, ed i processi C, D e I sul nodo 3: il traffico totale di rete è la somma degli archi attraversati dai tagli (le linee tratteggiate), quindi 30 unità. Nella Figura 8.25 (b) la partizione è diversa, e ha solo 28 unità di traffico di rete; se soddisfa tutti i vincoli di memoria e CPU, questa scelta è migliore, perché richiede meno comunicazione.

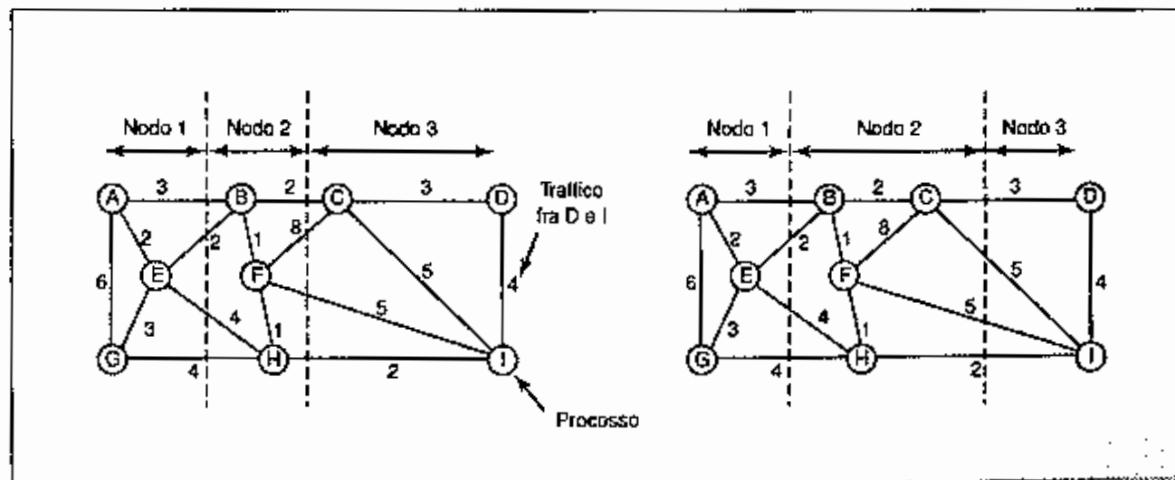


Figura 8.25 Due modi per allocare nove processi a tre nodi.

Intuitivamente, ciò che si sta facendo, è cercare cluster che siano strettamente accoppiati (con un flusso di traffico elevato all'interno dei cluster), ma che interagiscono poco con gli altri cluster (con un flusso di traffico fra cluster basso). Alcuni dei primi articoli dedicati all'argomento sono (Chow e Abraham, 1982; Lo, 1984; Stone e Bokhari, 1978).

## Un algoritmo euristico distribuito iniziato dal mittente

Consideriamo ora alcuni algoritmi distribuiti. Ad esempio, uno di questi algoritmi stabilisce che un processo, una volta creato, viene eseguito sul nodo che lo ha creato, sempre che esso non sia sovraccarico. La misura del sovraccarico potrebbe coinvolgere il numero eccessivo dei processi, la dimensione eccessiva del working set totale, o altre metriche; se il nodo è sovraccarico, seleziona casualmente un altro nodo, e gli chiede quale sia il suo carico (utilizzando la stessa metrica). Se il carico del nodo sondato è inferiore ad una certa soglia, il nuovo processo è inviato a quel nodo (Eager et al., 1986); in caso negativo, si sonda un'altra macchina. I tentativi non si ripetono all'infinito: se non si trovano macchine disponibili entro  $N$  tentativi, l'algoritmo termina, e il processo viene eseguito sulla macchina di partenza. L'idea è di permettere ai nodi sovraccarichi di provare a liberarsi del lavoro in eccesso, come vediamo nella Figura 8.26 (a).

Eager et al., (1986) hanno costruito un modello analitico di questo algoritmo basato sulla teoria delle code; esso stabilisce che l'algoritmo si comporta bene ed è stabile entro un ampio intervallo di parametri, che comprendono valori di soglia, costi di trasferimento e numero limite di tentativi diversi.

Tuttavia, occorre osservare che, in condizioni di carico pesante, tutte le macchine inviano costantemente richieste alle altre macchine, nel vano tentativo di trovarne una disponibile a svolgere altro lavoro. Pochi processi diminuiranno il proprio carico di lavoro, ma il sistema rischierà di incorrere in un considerevole overhead nel tentativo di farlo.

## Un algoritmo euristico distribuito iniziato dal ricevente

Un algoritmo complementare al precedente, è quello iniziato da un ricevente poco carico, come vediamo nella Figura 8.26 (b): quando un processo termina, il sistema controlla se abbia abbastanza lavoro, e in caso negativo, sceglie una macchina a caso e la

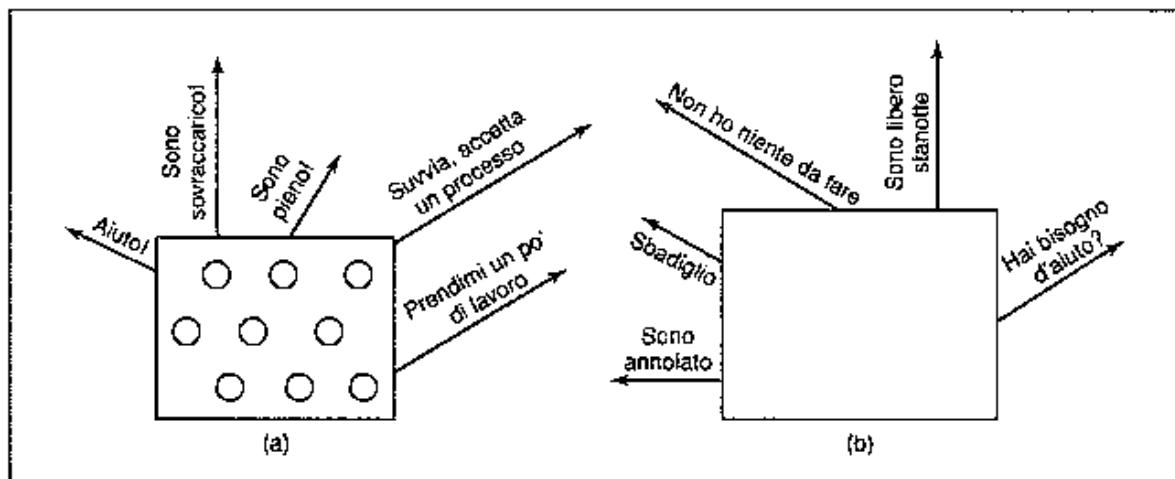


Figura 8.26 (a) Un nodo sovraccarico che cerca un nodo con minor carico per spedirgli processi. (b) Un nodo vuoto che cerca lavoro da svolgere.

interroga riguardo al suo carico di lavoro. Se quella macchina non ha nulla da offrire, vengono interrogate una seconda e una terza macchina; se non si trova lavoro disponibile entro N tentativi, il nodo cessa temporaneamente la ricerca, esegue il lavoro che ha accodato, e prova di nuovo quando il processo successivo termina. Se non è disponibile alcun lavoro, la macchina resta inattiva, e dopo un certo intervallo fisso di tempo inizia a provare di nuovo.

Un vantaggio di questo algoritmo è che non impone carichi aggiuntivi sul sistema nei momenti critici; l'algoritmo iniziato dal mittente fa un gran numero di tentativi proprio quando il sistema può tollerarli meno, cioè quando è pesantemente carico. Con l'algoritmo iniziato dal ricevente, quando il sistema è pesantemente carico, la possibilità che una macchina abbia lavoro insufficiente è bassa, e se ciò accade, sarà facile trovare lavoro da prendere. Naturalmente quando c'è poco lavoro da svolgere, l'algoritmo iniziato dal ricevente crea un considerevole traffico di richieste, poiché tutte le macchine disoccupate vanno a caccia di lavoro, ma è preferibile aver overhead quando il sistema è poco carico, piuttosto che nel caso contrario.

È anche possibile combinare entrambi questi algoritmi e avere macchine che cerchino di liberarsi di lavoro quando ne hanno troppo, e di acquisirne quando non ne hanno abbastanza. Inoltre le macchine possono migliorare rispetto alla ricerca casuale, tenendo traccia dei tentativi passati per determinare se alcune macchine siano costantemente sovraccaricate o poco cariche. Si potrebbe tentare prima con queste, se l'iniziatore stia cercando di scaricare lavoro, o di acquisirne.

### Un algoritmo basato sull'offerta

Un'altra classe di algoritmi cerca di trasformare il sistema di calcolo in un sistema economico in miniatura, con compratori e venditori di servizi, e prezzi determinati dalla domanda e dall'offerta (Ferguson et al., 1988). Gli attori chiave in questa economia sono i processi, che devono comprare tempo di CPU per eseguire i loro lavori, e i nodi, che vendono i loro cicli di CPU al miglior offerente.

Ciascun nodo rende noto il suo prezzo approssimato mettendolo in un file leggibile da tutti; tale prezzo non è fisso, ma indicativo del valore del servizio (effettivamente, è il prezzo pagato dall'ultimo cliente). Nodi diversi possono avere prezzi diversi, secondo la velocità, dimensione di memoria, presenza di hardware veloce in virgola mobile, e altre caratteristiche; si può anche pubblicare un'indicazione del servizio fornito, ad esempio il tempo di risposta aspettato.

Quando un processo vuole iniziare un processo figlio, si guarda attorno e controlla chi sta correntemente offrendo il servizio di cui ha bisogno, poi determina l'insieme dei nodi sui cui servizi può fare affidamento. Da questo insieme, calcola il candidato "migliore", dove migliore può significare il più economico, il più veloce o dotato del miglior rapporto prestazioni/prezzo, secondo l'applicazione; produce quindi un'offerta, più alta o più bassa del prezzo pubblicizzato, e la manda al primo nodo scelto.

I processori raccolgono tutte le offerte inviate loro, ed effettuano una scelta, presumibilmente scegliendo la più alta; informano i vincitori e i perdenti, ed eseguono il processo vincente. Il prezzo pubblicato del server è quindi aggiornato, in base al nuovo prezzo corrente.

Benché Ferguson et al. non entrino in dettagli, tale modello economico suscita una serie di interessanti domande, tra cui le seguenti. I processi da dove prendono il danaro per le offerte? Ricevono salari regolari? Ricevono tutti lo stesso salario mensile o i professori guadagnano più dei professori, che a loro volta guadagnano più degli studenti? Se vengono introdotti nuovi utenti nel sistema senza un corrispondente incremento delle

risorse, i prezzi saliranno (inflazione)? I nodi possono formare cartelli per sottrarre denaro agli utenti? Sono permessi sindacati di utenti? Si può addebitare anche lo spazio su disco? E come regolarsi con le stampanti? La stampa delle figure costerà di più rispetto a quella del testo, a causa dell'inchiostro o del toner utilizzati in più? L'elenco potrebbe proseguire a oltranza.

## 8.3 Sistemi distribuiti

Avendo completato il nostro studio sui multiprocessori e multicomputer, passiamo ora al terzo tipo di sistemi a processori multipli, i sistemi distribuiti. Questi sistemi sono simili ai multicomputer nel senso che ciascun nodo ha la sua memoria privata, senza memoria fisica condivisa nel sistema. I sistemi distribuiti sono anche più lasciamente accoppiati dei multicomputer.

In primo luogo, i nodi di un multicomputer generalmente hanno una CPU, una RAM e un'interfaccia di rete, e talvolta un hard disk per la paginazione; al contrario, ciascun nodo in un sistema distribuito è un computer completo, con una completa dotazione di periferiche. In secondo luogo, i nodi di un multicomputer si trovano normalmente in una stanza, in modo da poter comunicare attraverso una rete dedicata ad alta velocità, mentre i nodi di un sistema distribuito possono essere disseminati ovunque nel mondo. Infine, tutti i nodi di un multicomputer eseguono lo stesso sistema operativo, condividono un unico file system, ed appartengono ad una amministrazione comune, mentre i nodi di un sistema distribuito possono eseguire diversi sistemi operativi, ciascuno col proprio file system, ed appartenere ad amministrazioni diverse. Un tipico esempio di multicomputer è un sistema a 512 nodi in una stanza, in una industria o università che lavori ad esempio sulla modellazione farmaceutica; un tipico sistema distribuito si compone di migliaia di macchine lasciamente cooperanti su Internet. La Figura 8.27 confronta multiprocessori, multicomputer e sistemi distribuiti riguardo ai punti menzionati sopra.

Utilizzando questi criteri, i multicomputer si collocano a metà strada. È interessante chiedersi se i multicomputer siano più simili ai multiprocessori o ai sistemi distribuiti: può sembrare strano, ma la risposta dipende molto dal punto di vista. Sotto l'aspetto tecnico, i multiprocessori hanno memoria condivisa, a differenza degli altri due tipi di macchine, e tale differenza porta a concepire e modellare diversamente la programmazione. Invece, dal punto di vista dell'uso, multiprocessori e multicomputer sono rack di dispositivi in una stanza-macchine, che servono a risolvere problemi computazionalmente onerosi, mentre un sistema distribuito, che connette computer attraverso Internet, è mol-

| Problematica             | Multiprocessore         | Multicomputer                        | Sistema distribuito         |
|--------------------------|-------------------------|--------------------------------------|-----------------------------|
| Configurazione dei nodi  | CPU                     | CPU, RAM, interfaccia di rete        | Computer completo           |
| Periferiche dei nodi     | Tutte condivise         | Condivise, eccetto talvolta il disco | Insieme completo per nodo   |
| Ubicazione               | Nello stesso rack       | Nella stessa stanza                  | Anche in tutto il mondo     |
| Comunicazione fra i nodi | RAM condivisa           | Sistema di interconnessione dedicato | Rete tradizionale           |
| Sistemi operativi        | Uno, condiviso          | Multipli, dello stesso tipo          | Eventualmente tutti diversi |
| File system              | Uno, condiviso          | Uno, condiviso                       | Ciascun nodo ha il proprio  |
| Amministrazione          | Un'unica organizzazione | Un'unica organizzazione              | Organizzazioni diverse      |

Figura 8.27 Confronto di tre tipi di sistemi a CPU multipla.

to più impegnato nella comunicazione che nel calcolo, ed è usato in modo diverso.

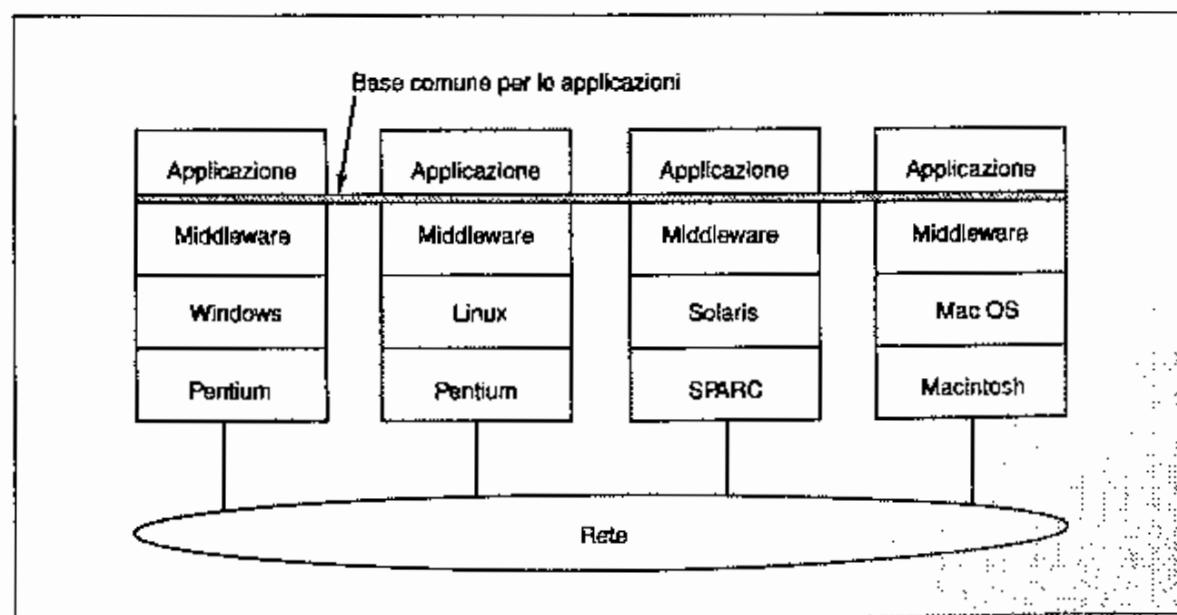
Per certi versi, l'accoppiamento debole dei computer in un sistema distribuito, è sia una forza, sia una debolezza: è una forza, perché è possibile usare i computer per una grande varietà di applicazioni, ma è una debolezza, perché programmare queste applicazioni è complesso, mancando un modello comune.

Le tipiche applicazioni Internet comprendono l'accesso a computer remoti (utilizzando *telnet* e *rlogin*), accesso ad informazione remota tramite il World Wide Web e FTP (File Transfer Protocol), la comunicazione da persona a persona (le *chat* e la posta elettronica), e molte applicazioni emergenti (ad esempio, il commercio elettronico, la telemedicina e l'apprendimento a distanza). Il problema comune di queste applicazioni è che ciascuna deve inventare ogni volta la ruota; ad esempio, la posta elettronica, FTP e World Wide Web, spostano tutti dei file da un punto *A* ad un punto *B*, ma ciascuno a suo modo, con convenzioni sulla terminologia, protocolli di trasferimento, tecniche di replicazione propri. Benché molti Web browser nascondano queste differenze all'utente medio, i meccanismi sottostanti sono completamente diversi. Nascondere questi aspetti a livello di interfaccia utente è come se un'agenzia di viaggi ci prenotasse un biglietto da Roma a Milano, e solo dopo scoprissimo se ci ha venduto un biglietto per un aereo, un treno o un bus.

Ciò che i sistemi distribuiti aggiungono alla rete sottostante è un paradigma (modello comune) che fornisce un modo uniforme di guardare all'intero sistema. L'intento del sistema distribuito è di trasformare un insieme debolmente connesso di macchine, in un sistema coerente basato su un concetto. Talvolta il paradigma è semplice, talvolta meno, ma l'idea è sempre fornire qualcosa che unifichi il sistema.

Un semplice esempio di paradigma unificante, in un contesto un poco diverso, lo troviamo in UNIX, dove tutti i dispositivi di I/O sono concepiti per apparire come file. Avere tastiere, stampanti e linee seriali tutte gestibili nello stesso modo, con le stesse primitive, rende più semplice interagire con esse, ed è preferibile al considerarle tutte concettualmente diverse.

Un modo perché un sistema distribuito ottenga una certa uniformità in presenza di hardware e sistemi operativi eterogenei, consiste in uno strato di software sopra il siste-



**Figura 8.28** Posizione del middleware in un sistema distribuito.

ma operativo. Questo strato, detto **middleware**, è illustrato nella Figura 8.28. Esso fornisce strutture dati e operazioni che permettono a processi e utenti su macchine lontane di interagire in modo coerente.

In un certo senso il middleware si può interpretare come il sistema operativo di un sistema distribuito; per questa ragione, se ne parla in un testo dedicato ai sistemi operativi. D'altro canto, non è un sistema operativo, perciò non scenderemo in dettaglio. Per una trattazione esauriente dei sistemi distribuiti, si veda *Distributed Systems* (Tanenbaum e Van Steen, 2002). Nella parte restante del capitolo vedremo rapidamente l'hardware di un sistema distribuito (la rete di computer sottostante), quindi il relativo software di comunicazione (i protocolli di rete). Quindi, vedremo alcuni paradigmi utilizzati in questi sistemi.

### 8.3.1 Hardware di rete

I sistemi distribuiti sono costruiti sopra a reti di computer, cui dedicheremo pertanto una breve introduzione. Esistono due principali tipi di reti: le **LAN** (Local Area Networks, reti locali), che coprono l'estensione di un edificio o di un campus, e le **WAN** (Wide Area Networks, reti geografiche), che possono estendersi ad una città, uno stato o anche al mondo intero. Il più importante tipo di LAN è Ethernet, che infatti prenderemo ad esempio. Per esemplificare WAN, prenderemo in esame Internet, anche se tecnicamente Internet non è una rete, bensì una federazione di migliaia di reti. Tuttavia, ai nostri fini, è sufficiente pensare ad essa come ad una WAN.

#### Ethernet

L'Ethernet classica descritta nello standard IEEE 802.3, si compone di un cavo coaxiale, cui sono collegati diversi computer. Il cavo è chiamato **Ethernet**, con riferimento all'*etere luminifero*, attraverso il quale un tempo si pensava si propagassero le radiazioni elettromagnetiche. Quando nel diciannovesimo secolo il fisico britannico James Clerk Maxwell scoprì che era possibile descrivere le radiazioni elettromagnetiche con un'equazione d'onda, gli scienziati ipotizzarono che lo spazio fosse pieno di un qualche mezzo etereo, dove si propagavano le radiazioni. Solo dopo il famoso esperimento di Michelson-Morley del 1887, che non riuscì a dimostrarne l'esistenza, i fisici capirono che la radiazione poteva propagarsi anche nel vuoto.

Nella prima versione di Ethernet, un computer era collegato al cavo facendo letteralmente un foro nel cavo stesso, e avvitandovi una connessione che conduceva al computer, chiamata **spina-vampiro**, rappresentata simbolicamente nella Figura 8.29 (a). Poiché

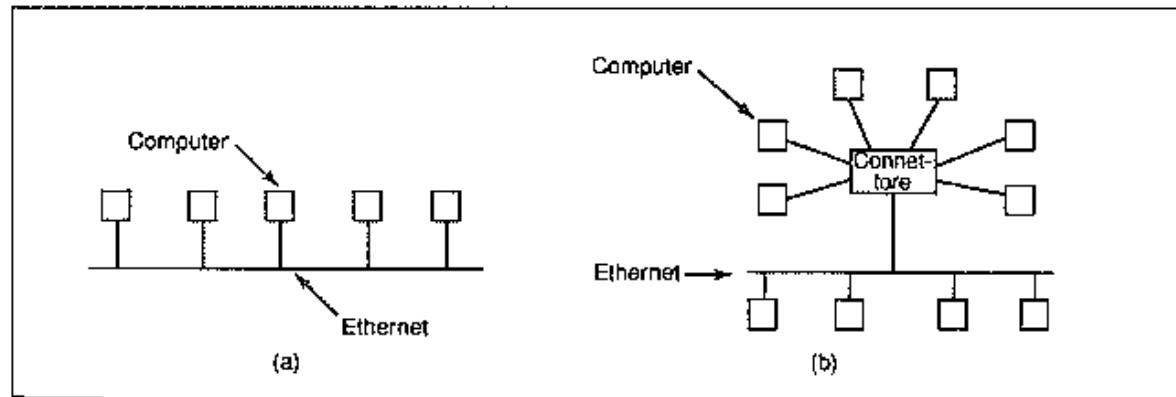


Figura 8.29 (a) Ethernet classico. (b) Ethernet con connettori.

era difficile sistemare correttamente tali spine, presto si utilizzarono connettori adatti, ma elettricamente, tutti i computer erano connessi come se i cavi della loro scheda di interfaccia fossero saldati insieme.

Per spedire un pacchetto su Ethernet, un computer dapprima ascolta il cavo per vedere se un altro computer sta trasmettendo in quel momento. In caso negativo, inizia a trasmettere un pacchetto, che si compone di una breve intestazione seguita da un messaggio vero e proprio di lunghezza variabile, da 0 a 1500 byte. Se il cavo è in uso, il computer semplicemente aspetta la fine della trasmissione per iniziare la propria.

Se due computer iniziano a trasmettere simultaneamente, si ha una collisione, rilevata da entrambi, che rispondono terminando la trasmissione, e aspettando per una quantità di tempo casuale fra 0 e  $T \mu s$ , quindi ripartono. Se avviene una seconda collisione, tutti i computer attendono per un tempo casuale compreso in un intervallo da 0 a  $2 T \mu s$ , quindi riprovano. Per ogni ulteriore collisione, il tempo massimo di attesa viene duplicato, riducendo la probabilità di nuove collisioni. Questo è noto come algoritmo di backoff esponenziale binario, che abbiamo visto in precedenza, per ridurre l'overhead del polling sui lock.

Ethernet ha una lunghezza massima del cavo, e un numero massimo di computer connessi. Per superare questi limiti, è possibile cablare con diverse reti Ethernet una grande costruzione o un campus, connessi attraverso dispositivi chiamati **bridge** o **ponte**. Un bridge permette al traffico di passare da un'Ethernet all'altra, quando la sorgente si trova in un punto, e la destinazione in un altro.

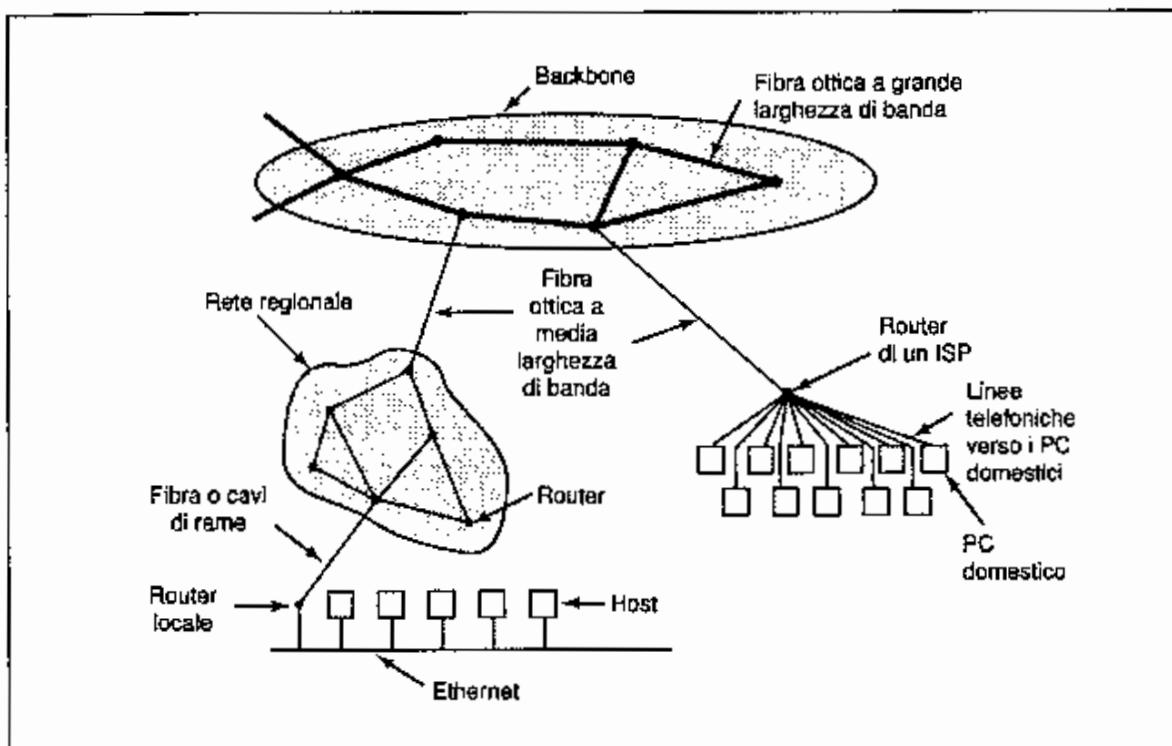
Per evitare collisioni, le reti Ethernet attuali utilizzano connettori, come vediamo nella Figura 8.29 (b). Ciascun connettore ha un certo numero di porte cui può essere connesso un computer, un'Ethernet oppure un altro connettore. Quando un pacchetto evita con successo tutte le collisioni, e si presenta ad un connettore, è ivi bufferizzato, e spedito alla porta cui è collegata la macchina di destinazione. Assegnando una porta a ciascun computer, è possibile eliminare tutte le collisioni, a prezzo di usare connettori più grandi. Sono anche possibili compromessi, con pochi computer per porta.

## Internet

Internet si è evoluta da ARPANET, una rete sperimentale a commutazione di pacchetto finanziata dall'Agenzia per i progetti di ricerca avanzati del Dipartimento della Difesa degli Stati Uniti. Ha visto la luce nel dicembre del 1969, quando ha connesso tre computer in California e uno nell'Utah; è stata progettata per essere una rete molto resistente ai guasti, in grado di continuare a trasmettere le comunicazioni militari anche in caso di impatto nucleare in alcune parti della rete, instradando automaticamente il traffico, per aggirare le macchine distrutte.

ARPANET crebbe rapidamente negli anni '70, e alla fine comprendeva centinaia di computer. In seguito, una rete radio a pacchetti, una rete satellitare e infine migliaia di reti Ethernet sono state collegate ad ARPANET, portando alla federazione di reti che ora conosciamo come Internet.

Internet si compone di due tipi di computer, host e router. Gli host sono PC, portatili, palmari, server, mainframe e altri computer, di privati o società che vogliono connettersi alla rete. I router, invece, sono computer specializzati per la connessione, che accettano pacchetti di ingresso provenienti da una o più linee, e che li rispediscono attraverso una delle loro molte linee di uscita. Un router è simile al connettore della Figura 8.29 (b), ma ne differisce per aspetti che non interessano la nostra trattazione. I router sono connessi insieme in grandi reti, e ciascuno ha cavi o fibre, diretti a molti altri router e host. Le società telefoniche e gli ISP (Internet Service Provider) gestiscono per i loro clienti gran-



**Figura 8.30 Una parte di Internet.**

di reti di router nazionali o mondiali.

La Figura 8.30 mostra una porzione di Internet. In alto abbiamo una delle backbone (letteralmente, spine dorsali), di norma gestite da un operatore di backbone, formata da diversi router, connessi da fibre ottiche ad ampia larghezza di banda, con connessioni ad altre backbone, gestite da altre compagnie telefoniche (in competizione). Di solito, nessun host si connette direttamente alla backbone, tranne le macchine per la gestione ed il test della backbone stessa, appartenenti alla compagnia telefonica.

Le reti regionali e i router degli ISP sono collegati alla backbone per mezzo di connessioni a fibre ottiche a media velocità; a loro volta reti Ethernet aziendali, ciascuna con un proprio router, sono connesse alla rete di router regionale, mentre i router degli ISP sono connessi a banchi di modem utilizzati dai loro clienti. Così, ogni host su Internet possiede un cammino, e spesso molti più, verso ogni altro host.

Tutto il traffico su Internet è spedito in forma di pacchetti, ciascuno dei quali contiene il proprio indirizzo di destinazione, utilizzato per l'instradamento. Quando un pacchetto arriva ad un router, esso estrae l'indirizzo di destinazione e ne cerca una parte in una tabella, per trovare a quale router e attraverso quale linea spedire il pacchetto. Questa procedura è ripetuta finché il pacchetto raggiunge l'host di destinazione. Le tabelle di instradamento sono molto dinamiche, e continuamente aggiornate, poiché i router ed i collegamenti possono attivarsi e disattivarsi, e le condizioni di traffico cambiano.

### 8.3.2 Servizi e protocollti di rete

Tutte le reti di computer forniscono servizi ai propri utenti (host e processi) e li implementano tramite regole per un corretto scambio di messaggi. Introduciamo ora questi argomenti.

## Servizi di rete

Le reti di computer forniscono servizi agli host e ai processi, che li usano. Un **servizio orientato alla connessione** è modellato sul sistema telefonico: per parlare a qualcuno, si solleva la cornetta, si compone il numero, si parla e quindi si riattacca. Allo stesso modo, per usare un servizio di rete orientato alla connessione, l'utente stabilisce prima una connessione, la usa e quindi la rilascia. L'aspetto essenziale di una connessione, è che funziona come un tubo: il mittente spedisce oggetti (cioè bit) da un estremo, e il ricevente li prende all'altro, nello stesso ordine.

Al contrario, un **servizio senza connessione** è modellato sul sistema postale: ciascun messaggio (lettera) contiene l'indirizzo di destinazione, e ciascuno è instradato attraverso il sistema indipendentemente da tutti gli altri, ma di norma quando due messaggi vengono spediti alla stessa destinazione, il primo spedito sarà anche il primo ad arrivare. In ogni caso, è possibile che il primo subisca dei ritardi, in modo che arrivi prima il secondo; questo non può accadere in un servizio orientato alla connessione.

Ciascun servizio è caratterizzato da una **qualità del servizio**. Alcuni servizi sono affidabili, nel senso che non perdono mai dati. Di solito, un servizio affidabile è implementato facendo in modo che il ricevente confermi la ricezione di ciascun messaggio spedendo indietro un apposito **pacchetto di riscontro o acknowledgement**, in modo che il mittente sappia che il suo messaggio è arrivato. I messaggi di riscontro introducono overhead e ritardi, necessari però per rilevare la perdita di pacchetti, anche se rallentano le operazioni.

Una situazione tipica in cui un servizio orientato alla connessione è appropriato, è il trasferimento di file. Il possessore del file vuol essere certo che tutti i bit arrivino correttamente e nello stesso ordine di invio; ben pochi preferirebbero un servizio che talvolta cambiasse questo ordine, o perdesse alcuni bit, nemmeno se fosse molto più veloce.

Esistono due varianti minori di servizio affidabile orientato alla connessione: sequenze di messaggi e flussi di byte. Nel primo caso, i confini dei messaggi sono preservati: quando si spediscono due messaggi da 1 KB, arrivano due distinti messaggi da 1 KB, e mai uno solo da 2 KB. Nel secondo caso, la connessione è semplicemente un flusso di byte, senza confini fra i messaggi: quando arrivano al ricevente 2 KB, non è possibile stabilire se siano stati inviati come un messaggio da 2 KB, 2 messaggi da 1 KB o 2048 messaggi da 1 byte. Se spediamo in rete alla tipografia le pagine di un libro come messaggi separati, potrebbe essere importante conservare i confini tra i messaggi. D'altro canto, con un terminale che effettua il login in un sistema timesharing remoto, un flusso di byte dal terminale al computer è tutto ciò che serve.

Per alcune applicazioni, i ritardi introdotti dai messaggi di riscontro sono inaccettabili; una di queste è la trasmissione di voci digitalizzate. Un utente telefonico preferisce sentire un poco di rumore sulla linea, o una parola confusa di tanto in tanto, piuttosto che aspettare il messaggio di riscontro.

Non tutte le applicazioni richiedono connessioni. Ad esempio, per testare la rete occorre soltanto poter spedire un unico pacchetto, che abbia una buona probabilità di arrivare, anche se non garantita. Un servizio senza connessione non affidabile (cioè senza messaggi di riscontro) è spesso definito **servizio datagramma**, per analogia coi telegrammi, che ugualmente non forniscono un messaggio di riscontro al mittente.

In altre situazioni, non si vuol stabilire una connessione per spedire un messaggio breve, ma l'affidabilità resta fondamentale; per queste applicazioni, si può fornire un **servizio datagramma con messaggio di riscontro**, come se si spedisce una raccomandata con ricevuta di ritorno. Quando la ricevuta torna indietro, il mittente è assolutamente certo che la lettera è arrivata al destinatario, senza perdersi per strada.

Un altro servizio è quello di **richiesta-risposta**. Il mittente trasmette un singolo dato-

|                            | Servizio                      | Esempio                          |
|----------------------------|-------------------------------|----------------------------------|
| Orientato alla connessione | Flusso di messaggi affidabile | Serie di pagine di un libro      |
|                            | Flusso di byte affidabile     | Login remota                     |
| Senza connessione          | Connessione non affidabile    | Voce digitalizzata               |
|                            | Datagramma non affidabile     | Pacchetti per il test della rete |
|                            | Datagramma con riscontro      | Posta raccomandata               |
|                            | Richiesta-risposta            | Interrogazione di un database    |

Figura 8.31 Sei diversi tipi di servizi di rete.

gramma, che contiene una richiesta, e il messaggio di ritorno contiene la risposta. Ad esempio, ricade in questa tipologia una richiesta ad una biblioteca locale, per sapere dove si parla l'uighur. Il servizio richiesta-risposta è comunemente usato per implementare la comunicazione nel modello client-server: il client formula una richiesta, e il server gli risponde. La Figura 8.31 riassume i tipi di servizi sopra esaminati.

## Protocolli di rete

Tutte le reti hanno regole peculiari, per stabilire quali messaggi spedire, e quali risposte dare; ad esempio, in certi casi (come un trasferimento di file), quando un messaggio è spedito da una sorgente ad una destinazione, si richiede che quest'ultima mandi un messaggio di riscontro, per attestare la corretta ricezione. In altri casi (ad esempio la telefonia digitale) noi ci si aspetta questo riscontro. L'insieme di regole secondo le quali comunicano particolari computer, è definita **protocollo**. Ne esistono numerosi, ad esempio i protocolli router-router, quelli host-host, ed altri ancora; una trattazione completa delle reti di computer e dei loro protocolli si trova in *Reti di computer* (Tanenbaum 1996).

Tutte le reti moderne utilizzano un cosiddetto **stack di protocolli**, per impilare uno sull'altro diversi protocolli. In ciascuno strato si affrontano problemi diversi; ad esempio, a livello inferiore i protocolli specificano come definire l'inizio e la fine di un pacchetto all'interno di un flusso di bit. Ad un livello più alto, i protocolli trattano come instradare pacchetti, attraverso reti complesse, dalla sorgente alla destinazione, ed a livello ancora superiore, si fanno carico di garantire che tutti i pacchetti, in un messaggio a più pacchetti, siano arrivati privi di errori, e nell'ordine corretto.

Poiché la maggior parte dei sistemi distribuiti usa Internet, i loro protocolli base sono i due principali protocolli Internet: IP e TCP. IP (**Internet Protocol**) è un protocollo datagramma, per mezzo del quale un mittente inserisce un datagramma di dimensioni fino a 64 KB nella rete, e spera che arrivi: non si danno garanzie. Il datagramma si può frammentare in pacchetti più piccoli, man mano che passa attraverso Internet, e tali pacchetti viaggiano indipendentemente, seguendo anche cammini diversi. Quando tutti i pezzi arrivano a destinazione, sono assemblati nell'ordine corretto e recapitati.

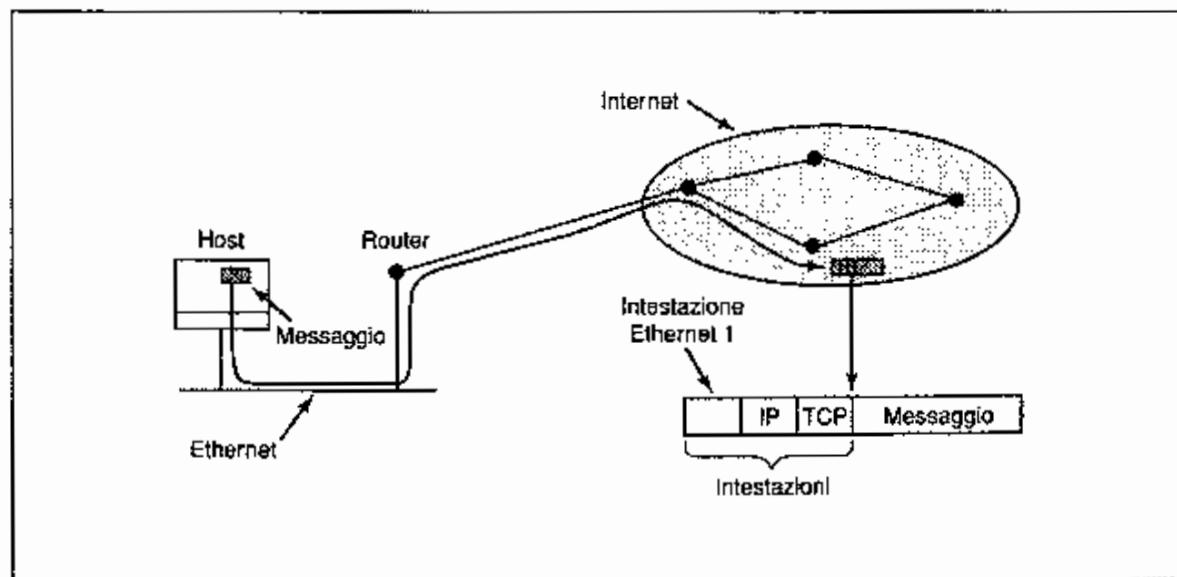
Attualmente, si usano due versioni di IP, la v4 e la v6; descriveremo v4 perché è ancora la più diffusa, anche se v6 si sta affermando. Ciascun pacchetto v4 inizia con un'intestazione di 40 byte, che contiene, fra gli altri campi, un indirizzo a 32 bit della sorgente, e

uno a 32 bit della destinazione. Sono chiamati **indirizzi IP**, e formano la base dell'instradamento in Internet. Per convenzione, sono scritti come quattro decimali compresi nell'intervallo 0-255, separati da punti, come ad esempio in 192.31.231.65. Quando gli arriva un pacchetto, il router estrae l'indirizzo IP di destinazione, e se ne serve per instradare il pacchetto.

Poiché i datagramma IP sono privi di messaggio di riscontro, IP da solo non è sufficiente per una comunicazione affidabile in Internet. Per ottenerla, di solito si pone sopra IP un altro protocollo, TCP (Transmission Control Protocol); TCP usa IP per fornire flussi orientati alla connessione. Per utilizzare TCP, un processo per prima cosa stabilisce una connessione ad un processo remoto. Il processo richiesto è specificato da un indirizzo IP di una macchina, e da un numero di porta su quella macchina, su cui sono in ascolto i processi interessati a ricevere connessioni in ingresso. Stabilita la connessione, il processo continua attraverso di essa a trasmettere byte, il cui arrivo è garantito senza errori, e nell'ordine corretto. L'implementazione TCP ottiene questa garanzia utilizzando numeri di sequenza, checksum e ritrasmissione di pacchetti non correttamente ricevuti. Tutto ciò è trasparente ai processi mittente e ricevente: sembra proprio una comunicazione affidabile fra processi, come una pipe UNIX.

Per vedere come tutti questi protocolli interagiscano, si consideri il semplice caso di un messaggio molto breve, che non debba essere frammentato a nessun livello, quando l'host si trova su una connessione Ethernet a Internet. Cosa accade di preciso? Il processo utente genera il messaggio, ed effettua una chiamata di sistema per spedirlo attraverso una connessione TCP stabilita in precedenza. Lo stack di protocolli del kernel, aggiunge un'intestazione TCP, e poi una IP all'inizio, quindi passa al driver Ethernet, il quale aggiunge un'intestazione Ethernet, che dirige il pacchetto al router della rete. Questo router inserisce quindi il pacchetto in Internet, come vediamo nella Figura 8.32.

Per stabilire una connessione con un host remoto (o anche per spedirgli un datagramma), occorre conoscere il suo indirizzo IP. Poiché gestire liste di indirizzi IP a 32 bit non è conveniente, è stato inventato uno schema chiamato DNS (Domain Name System), cioè un database che mappa i nomi ASCII degli host nei loro indirizzi IP; così è possibile utilizzare il nome DNS *star.cs.vu.nl* invece del corrispondente indirizzo IP 130.37.24.6. I nomi DNS sono ampiamente noti, perché gli indirizzi di posta elettronica Internet sono della



**Figura 8.32** Accumulazione delle intestazioni di un pacchetto.

forma `nome-utente@DNS-nome-host`. Questo sistema di nomi permette al programma di posta sull'host del mittente di consultare l'indirizzo IP dell'host di destinazione nel database DNS, di stabilire una connessione TCP col processo demone della posta sull'host remoto, e di spedire il messaggio come un file. Il `nome-utente` è spedito per identificare in quale casella porre il messaggio.

### 8.3.3 Middleware basato sui documenti

Ora che abbiamo acquisito una certa conoscenza di base sulle reti e i protocolli, possiamo passare ai diversi strati middleware che possono sovrapporsi alla rete base, per produrre un paradigma consistente per le applicazioni e gli utenti. Partiamo da un esempio semplice, ma ben noto: il World Wide Web. Il Web è stato inventato nel 1989 da Tim Berners-Lee al CERN, il Centro di Ricerca europeo di fisica nucleare, e da allora si è diffuso a macchia d'olio in tutto il mondo.

Il paradigma originale dietro al Web era molto semplice. Ciascun computer può contenere uno o più documenti, chiamati **pagine Web**, ciascuna delle quali contiene testo, immagini, icone, suoni, film eccetera, oltre a **link ipertestuali** o **hyperlink** (puntatori) ad altre pagine Web. Quando un utente richiede una pagina Web utilizzando un programma chiamato **Web browser**, la pagina è visualizzata sullo schermo; fare clic su un link provoca la sostituzione della pagina corrente sullo schermo con la pagina selezionata. Benché fronzoli e orpelli vari siano stati di recente aggiunti sul Web, il paradigma sottostante è ancora chiaramente presente: il Web è un grande grafo diretto di documenti che possono puntare ad altri documenti, come vediamo nella Figura 8.33.

Ciascuna pagina Web ha un unico indirizzo, chiamato **URL (Uniform Resource Locator)**, della forma `protocollo://nome-DNS/nome-file`; il protocollo più comune è `http` (Hyper-text Transfer Protocol), ma esistono anche `ftp` ed altri. Quindi viene il nome DNS dell'host che contiene il file, infine un nome di file locale, che specifica di quale file si abbia bisogno.

Tutto il sistema è organizzato così: il Web è fondamentalmente un sistema client-server, con l'utente che rappresenta il client, e il sito Web che rappresenta il server. Quando l'utente dà al browser un URL, o scrivendola per esteso, o facendo un click su di un link ipertestuale nella pagina corrente, il browser compie una serie di passi per prelevare la

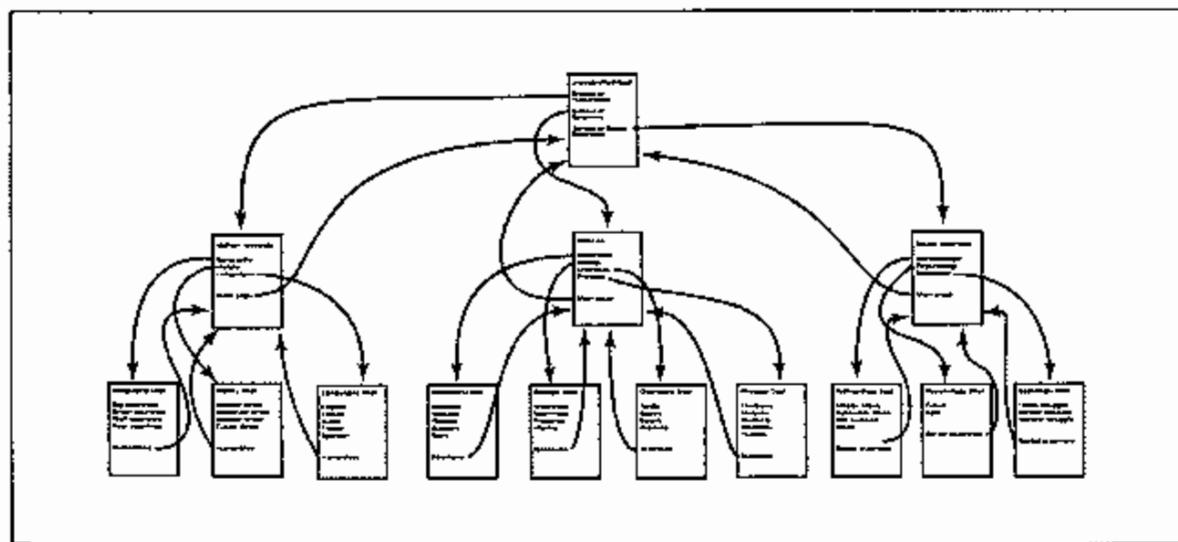


Figura 8.33 Il Web è un grande grafo diretto di documenti.

pagina Web richiesta. Ad esempio, si supponga che l'URL fornito sia <http://www.acm.org/dl/faq.html>; il browser effettua i seguenti passi per ottenere la pagina.

1. Il browser interroga il DNS per ottenere l'indirizzo IP di *www.acm.org*.
2. Il DNS risponde con 199.22.69.151.
3. Il browser fa una connessione TCP alla porta 80 dell'indirizzo 199.22.69.151.
4. Richiede quindi il file *dl/faq.html*.
5. Il server *www.acm.org* manda il file *dl/faq.html*.
6. La connessione TCP è rilasciata
7. Il browser visualizza tutto il testo contenuto in *dl/faq.html*.
8. Il browser preleva e visualizza tutte le immagini contenute in *dl/faq.html*.

Approssimativamente, questa è la base del Web e del suo funzionamento. Molte altre caratteristiche sono state aggiunte al Web base, tra cui le pagine di stile, le pagine Web dinamiche generate al momento, le pagine Web contenenti piccoli programmi o script eseguiti sulla macchina client ed altro, ma questi elementi esulano dalla nostra trattazione.

### 8.3.4 Middleware basato sul file system

L'idea alla base del Web, è far sembrare un sistema distribuito come una gigantesca collezione di documenti ipertestuali; il secondo passo consiste nel rappresentare un sistema distribuito come un grande file system. In questa sezione prenderemo in esame alcuni dei problemi relativi alla progettazione di un file system esteso a tutto il mondo.

Utilizzare un modello di file system per un sistema distribuito, significa avere un solo file system globale, con gli utenti di tutto il mondo in grado di leggere e scrivere file, secondo le specifiche autorizzazioni di cui dispongono. La comunicazione si ottiene attraverso processi che scrivono dati in un file, ed altri che li leggono; in questo contesto, sorgono molte problematiche tipiche dei file system standard, ed anche altre, inerenti ai sistemi distribuiti.

#### Modello di trasferimento

Il primo problema è scegliere fra il modello **upload/download** e il **modello di accesso remoto**. Nel primo caso, mostrato nella Figura 8.34 (a), un processo accede ad un file copiandolo prima dal server remoto su cui risiede. Quando il file viene letto o scritto, lo si fa localmente per aumentare le prestazioni; ed alla fine del processo, se il file è stato aggiornato, è rispedito al server. Con il modello di accesso remoto, il file rimane sul server, e il client spedisce comandi perché siano eseguiti in remoto, come vediamo nella Figura 8.34 (b).

I vantaggi del modello upload / download sono la semplicità e il fatto che trasferire interi file in una volta sola è più efficiente che farlo in piccole parti, tra gli svantaggi, ci vuole abbastanza memoria locale per l'intero file, c'è un eventuale spreco di risorse per spostare tutto il file, se ne servono solo alcune parti, e ci potrebbero essere problemi di consistenza, se ci sono più utenti in contemporanea.

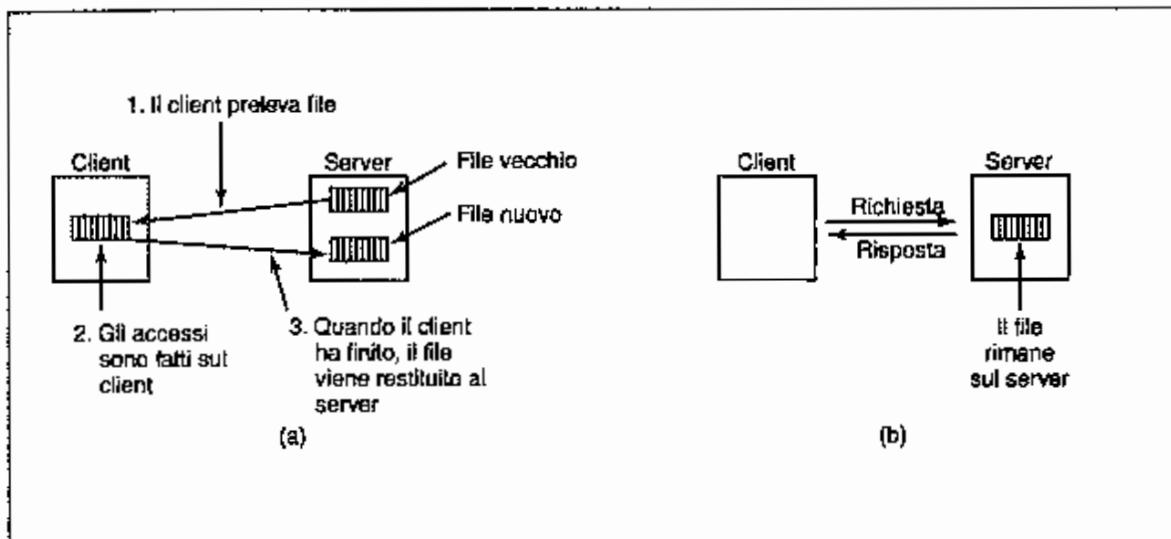


Figura 8.34 (a) Il modello upload/download. (b) Il modello di accesso remoto.

## La gerarchia delle directory

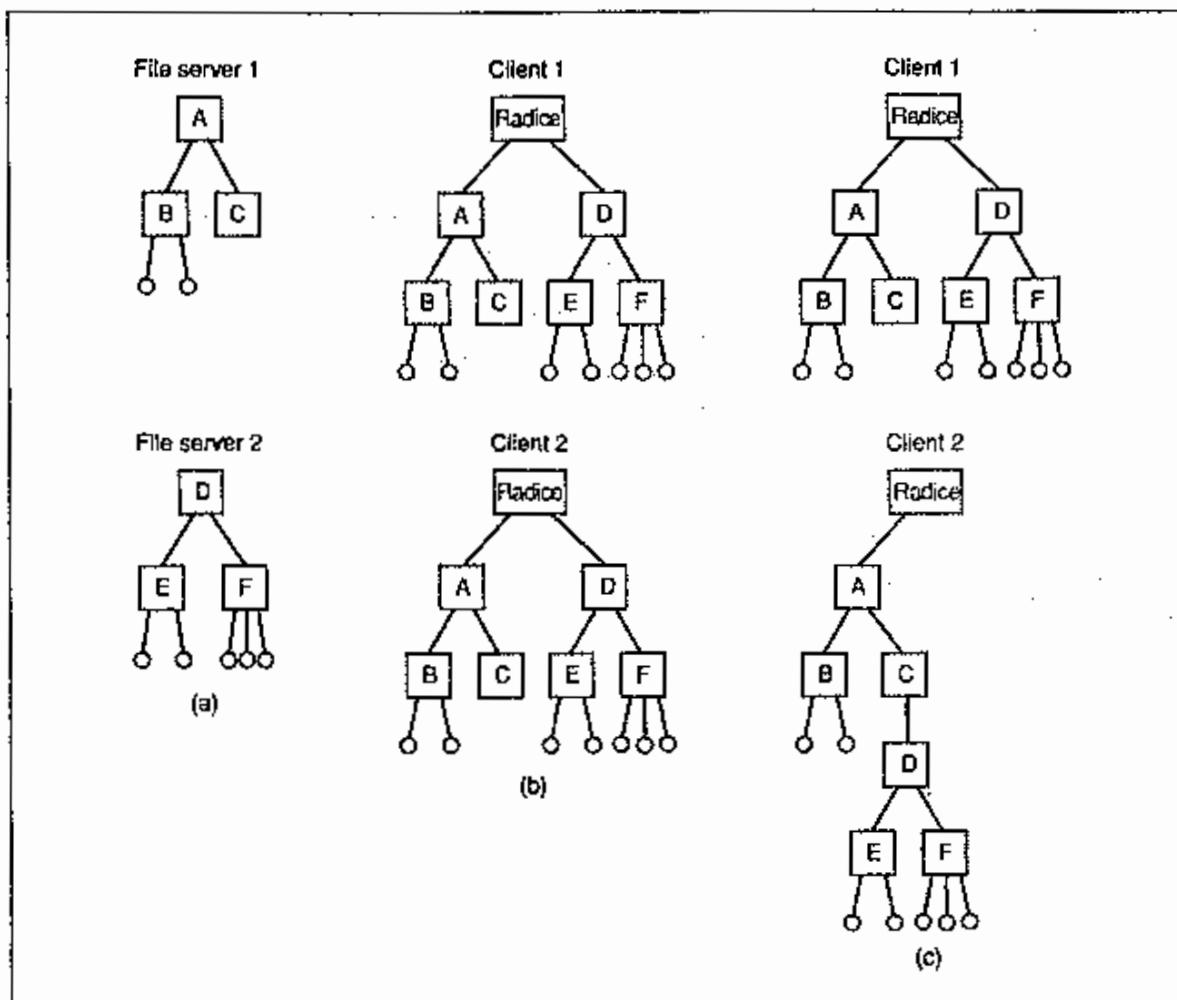
I file costituiscono un aspetto del problema; l'altro è il sistema delle directory. Tutti i file system distribuiti supportano directory che contengono più file; il problema di progettazione successivo è decidere se tutti i client abbiano la stessa vista della gerarchia delle directory. Come esempio, si consideri la Figura 8.35. Nella Figura 8.35 (a) vediamo due file server, ciascuno con tre directory e alcuni file. Nella Figura 8.35 (b) abbiamo un sistema in cui tutti i client (e le altre macchine) hanno la stessa vista del file system distribuito. Se il cammino /D/E/x è valido su una macchina, lo è anche su tutte le altre.

Al contrario, nella Figura 8.35 (c) macchine differenti possono avere viste differenti del file system: riprendendo l'esempio precedente, il cammino /D/E/x potrebbe essere valido per il client 1, ma non per il 2. In sistemi che gestiscono diversi file server attraverso il mount remoto, la Figura 8.35 (c) è la norma; tale schema è flessibile e semplice da implementare, ma ha lo svantaggio di non far sì che l'intero sistema si comporti come un unico sistema timesharing vecchio stampo. In un sistema timesharing, infatti, il file system appare uguale ad ogni processo [in altre parole il modello della Figura 8.35 (b)], e questa proprietà rende il sistema facile da capire e da programmare.

Un problema strettamente correlato è se esista o no una directory radice globale, che tutte le macchine riconoscano appunto come radice. Un modo per ottenere una directory radice globale consiste nell'avere una radice che contenga un'entry per ciascun server e null'altro; i cammini prendono la forma /server/cammino, il che ha i suoi svantaggi, ma almeno è lo stesso ovunque nel sistema.

## Trasparenza dei nomi

Il problema principale con questa forma di gestione dei nomi è la limitata trasparenza; in questo contesto è opportuno distinguere due forme di trasparenza, molto rilevanti. La prima, **trasparenza della localizzazione**, significa che il nome del cammino non deve dare suggerimenti rispetto a dove sono localizzati i file. Un cammino come /server1/dir1/dir2/x rende noto a tutti che x si trova sul server 1, ma non dove sia localizzato tale server; il server è libero di muoversi ovunque nella rete, senza che il nome del



**Figura 8.35** (a) Due file server. I quadrati sono le directory ed i cerchi rappresentano i file. (b) Un sistema in cui tutti i client hanno la stessa vista del file system. (c) Un sistema in cui client diversi possono avere viste diverse del file system.

cammino debba essere cambiato. Quindi, questo sistema dispone di trasparenza della localizzazione.

Comunque, si supponga che il file *x* sia molto grande e che ci sia poco spazio sul server 1, ma ampia disponibilità di spazio sul server 2; il sistema potrebbe voler spostare automaticamente il file *x* sul server 2. Tuttavia, quando il primo componente di tutti i nomi dei cammini è il server, il sistema non può spostare automaticamente il file in un altro server, anche se *dir1* e *dir2* esistono su entrambi i server. Il problema è che lo spostamento automatico dei file cambia il nome del cammino da */server1/dir1/dir2/x* a */server2/dir1/dir2/x*, ed i programmi che hanno la prima stringa incorporata, cesseranno di funzionare. I sistemi in cui i file possono essere spostati senza che il loro nome cambi sono dotati di **indipendenza della localizzazione**; un sistema distribuito che incorpore i nomi delle macchine o dei server nei nomi dei cammini, chiaramente non è indipendente nella localizzazione. Non lo è neppure un sistema basato sul *mount remoto*, poiché non è possibile spostare un file da un gruppo (l'unità di *mount*) ad un altro, ed essere ancora in grado di utilizzare il vecchio nome del cammino. L'indipendenza dalla localizzazione non è facile da ottenere, ma è una proprietà desiderabile per un sistema distribuito.

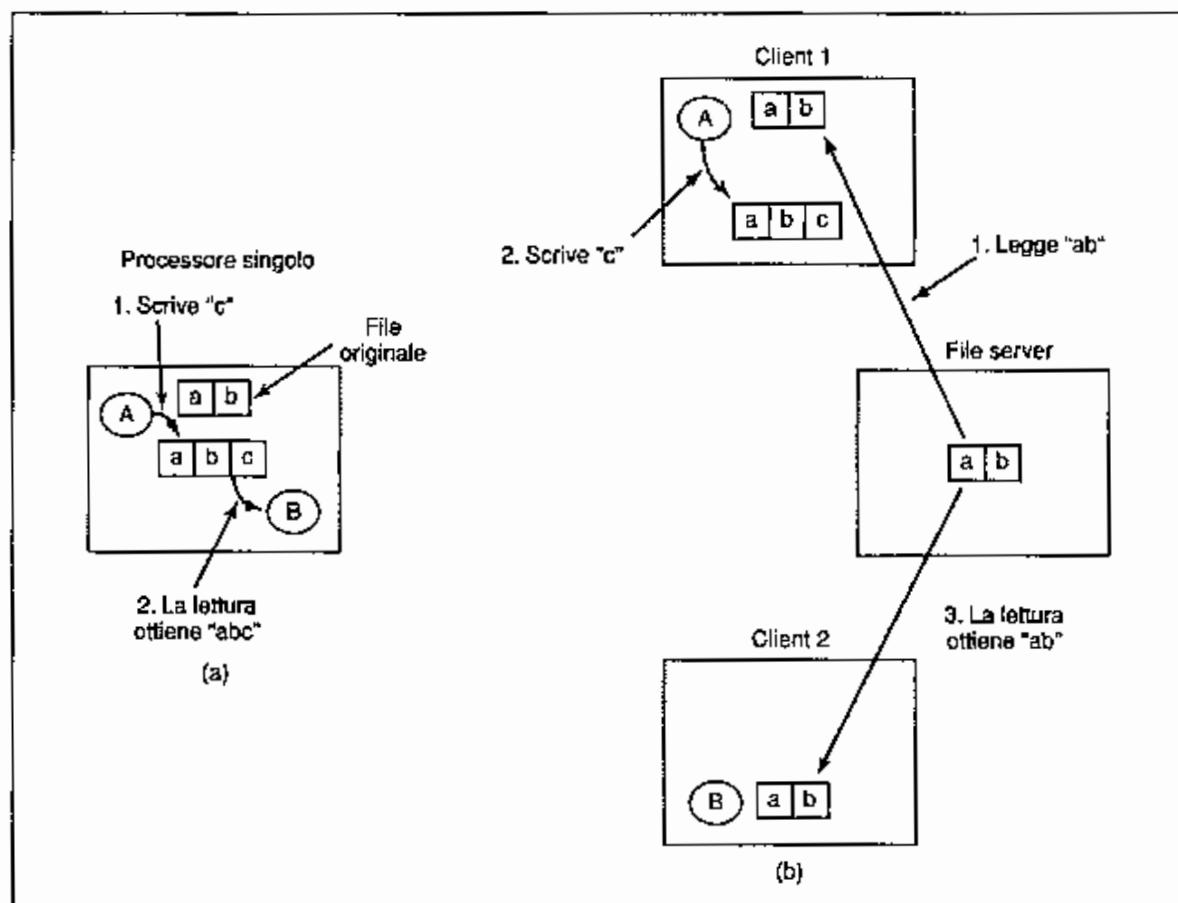
In sintesi, esistono tre approcci comuni alla gestione dei nomi in un sistema distribuito:

1. Gestione dei nomi macchina + cammino, come */macchina/cammino* oppure *macchina:cammino*.
2. Mount dei file system remoti nella gerarchia locale dei file.
3. Un singolo spazio dei nomi che appare uguale su tutte le macchine.

I primi due sono facili da implementare, specialmente per connettere sistemi esistenti che non sono stati progettati per un utilizzo distribuito; l'ultimo è difficile e richiede un'attenta progettazione, ma semplifica la vita ai programmati ed agli utenti.

## Semantica della condivisione dei file

Quando due o più utenti condividono lo stesso file, è necessario definire precisamente la semantica della lettura e della scrittura per evitare problemi. In sistemi mono-processore, di norma la semantica afferma che, quando una chiamata di sistema `read` segue una `write`, la `read` deve restituire il valore appena scritto, come mostrato in Figura 8.36 (a). Allo stesso modo, quando due `write` avvengono in rapida successione, seguite da una `read`, il valore letto è quello memorizzato dall'ultima `write`. In effetti, il sistema



**Figura 8.36** (a) Consistenza sequenziale. (b) In un sistema distribuito con cache la lettura di un file può restituire un valore obsoleta.

applica un ordinamento su tutte le chiamate di sistema, e tutti i processori vedono lo stesso ordinamento; chiameremo questa proprietà **consistenza sequenziale**.

In un sistema distribuito, si può ottenerla facilmente quando è presente un solo file server e i client non memorizzano localmente i file. Tutte le `read` e le `write` vanno direttamente al file server, che le elabora in modo strettamente sequenziale.

In pratica, le prestazioni di un sistema distribuito in cui tutte le richieste relative ai file devono passare attraverso un unico server, spesso sono scarse. Questo problema è risolto permettendo ai client di mantenere copie locali dei file particolarmente usati, nelle loro cache private. Però, se il client 1 modifica un file nella cache locale, e poco dopo il client 2 lo legge dal server, otterrà un file obsoleto, come vediamo nella Figura 8.36 (b).

Una via d'uscita consiste nel propagare immediatamente al server tutti i cambiamenti ai file che si trovano nella cache; benché concettualmente semplice, l'approccio è inefficiente. Una soluzione consiste nel rendere meno stretta la semantica della condivisione dei file: invece di richiedere che una `read` veda gli effetti di tutte le `write` precedenti, si potrebbe formulare una nuova regola che dica: "I cambiamenti ad un file aperto sono visibili inizialmente solo al processo che li ha generati. Solo quando il file viene chiuso, i cambiamenti diventano visibili per gli altri processi". L'adozione di una tale regola, non cambia ciò che avviene nella Figura 8.36 (b), ma ridefinisce il comportamento effettivo (B ottiene il valore originale del file) come corretto. Quando il client 1 chiude il file, invia una copia al server, in modo che le `read` seguenti ricevano il nuovo valore, come richiesto. In effetti, questo è il modello upload/download della Figura 8.34; questa regola semantica è ampiamente implementata, ed è nota come **semantica di sessione**.

La semantica di sessione pone il problema di cosa possa accadere, se due o più client stessero contemporaneamente mantenendo nella cache e modificando lo stesso file. Una soluzione è che, alla chiusura, ciascun file sia rispedito al server, in modo che il risultato finale dipenda da quello che viene chiuso per ultimo. Un'alternativa meno gradevole, ma un po' più facile da implementare, consiste nel dire che il risultato finale è uno dei candidati, senza specificare quale.

Un approccio alternativo alla semantica di sessione è utilizzare il modello upload/download, ponendo automaticamente il lock ad un file che è stato scaricato. I tentativi di scaricare il file da parte degli altri client saranno tenuti in coda, finché il primo client restituisce il file su cui ha posto il lock. Se c'è una richiesta sostenuta per un file, il server potrebbe mandare dei messaggi al client che lo detiene, chiedendogli di sbrigarsi, ma non è detto che ciò sia d'aiuto. Tutto sommato, mantenere corretta la semantica dei file condivisi è un problema complesso, privo di soluzioni eleganti ed efficienti.

## AFS

Sono stati costruiti ed impiegati diversi sistemi middleware, basati sul file system. Ne discuteremo uno (AFS), basato sul modello upload/download della Figura 8.34 (a). Nel Capitolo 10 ne discuteremo un secondo, NFS, basato sul modello di accesso remoto della Figura 8.34 (b).

AFS è stato progettato e implementato all'Università Carnegie-Mellon (Howard et al., 1988; Morris et al., 1986; Satyanarayanan 1985); in origine, si chiamava Andrew File System, in onore dei primi beneficiari dell'Università, Andrew Carnegie e Andrew Mellon. L'obiettivo del progetto, partito nei primi anni '80, era fornire agli studenti e ai docenti della CMU una potente workstation individuale in ambiente UNIX, ma con un file system condiviso. Qui il file system è stato utilizzato come middleware per connettere un insieme di workstation in un sistema coerente.

Ciascun utente AFS ha un workstation privata, che esegue una versione di UNIX leg-

germente modificata. Le modifiche consistono nell'aggiunta al kernel di una porzione di codice chiamata *venus*, e nell'eseguire un file server, chiamato *vice* nello spazio utente (in origine, anche *venus* era eseguita nello spazio utente, quindi fu spostata nel kernel per migliorare le prestazioni). Le posizioni di *venus* e *vice* le vediamo nella Figura 8.37 (a). Le workstation degli utenti sono raggruppate in cellule, per fini amministrativi; una cellula potrebbe essere una LAN, oppure una serie di LAN interconnesse, o anche un intero dipartimento universitario.

Lo spazio dei nomi visibile ai programmi utente appare come un albero UNIX tradizionale, con l'aggiunta delle directory */cmu* e */cache*, come illustrato nella Figura 8.37 (b). La directory */cache* contiene file remoti posti nella cache; la directory */cmu* contiene i nomi delle cellule remote condivise, sotto alle quali giacciono i rispettivi file system. In effetti, i file system remoti sono montati in */cmu*; le altre directory e gli altri file sono strettamente locali, e non condivisi. Sono permessi link simbolici, dai nomi locali dei file ai file condivisi, come indicato da *sh* in Figura 8.37 (b).

L'idea essenziale di AFS consiste nel fare in modo che ciascun utente lavori il più possibile localmente e interagisca il meno possibile con il resto del sistema. Quando viene aperto un file, il codice di *venus* cattura la chiamata *open* e scarica l'intero file (o un'ampia porzione se il file è molto grande) nel disco locale, e lo inserisce nella directory */cache*. Il descrittore di file restituito da una chiamata *open* si riferisce al file in */cache*, cosicché le successive chiamate *read* e *write* utilizzano il file nella cache.

La semantica offerta da AFS è simile a quella di sessione. Quando un file è aperto, è prelevato dal server appropriato, e posto in */cache*, sul disco locale della workstation; tutte le *read* e le *write* operano sulla copia nella cache. Quando il file viene chiuso, è rimmorizzato sul server.

In ogni caso, per evitare che altri processi utilizzino inconsapevolmente file vecchi, quando ciò causerebbe dei problemi, se *venus* scarica un file nella propria cache, comunica a *vice* se debba o no prestare attenzione alle successive *open* dei processi delle altre workstation. In caso affermativo, *vice* registra la posizione del file posto nella cache, e se altrove nel sistema un altro processo apre il file, *vice* manda un messaggio a *venus* chiedendo di invalidare la propria entry nella cache, e di restituire la copia se è modificata.

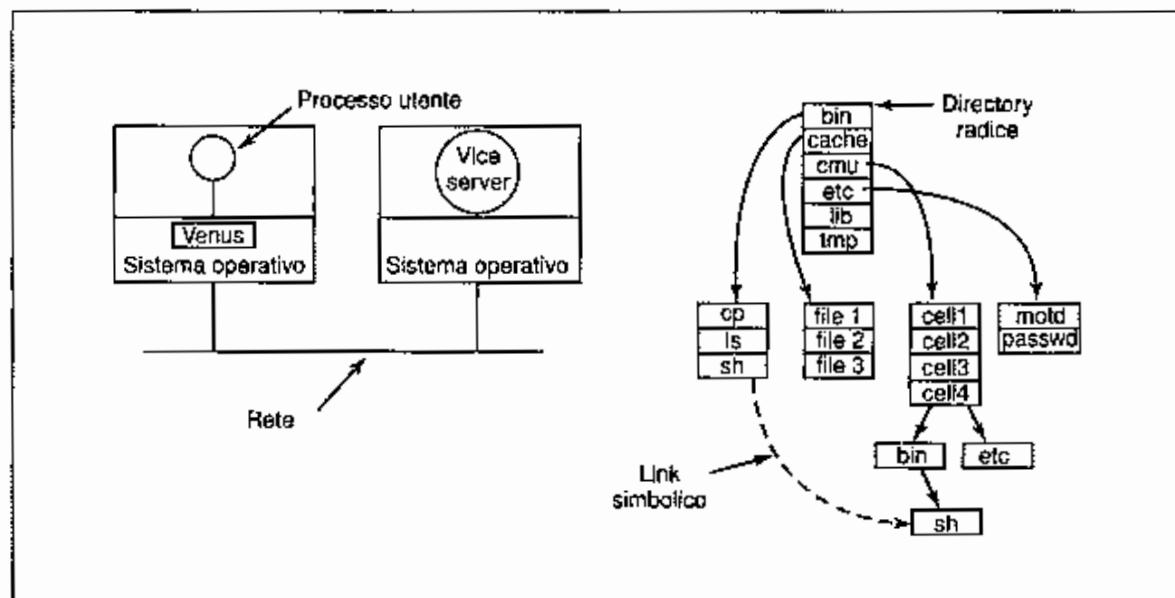


Figura 8.37 (a) La posizione di *venus* e *vice* in AFS. (b) Una vista del file system da parte di un client.

### 8.3.5 Middleware basato su oggetti condivisi

Consideriamo ora un terzo paradigma: invece di affermare che tutto è un documento, o un file, affermeremo che tutto è un **oggetto**. Un oggetto è una collezione di variabili, associate ad un insieme di procedure d'accesso, chiamate **metodi**. Ai processi non è consentito accedere direttamente alle variabili, ma si richiede loro di chiamare dei metodi.

#### CORBA

Alcuni linguaggi di programmazione come C++ e Java, sono orientati agli oggetti, intesi come tali piuttosto a livello di linguaggio che come oggetti a tempo di esecuzione. Un noto sistema basato su oggetti a tempo di esecuzione è **CORBA** (Common Object Request Broker Architecture) (Vinosky 1997). CORBA è un sistema client-server, dove i processi client sulle macchine client possono chiamare operazioni su oggetti localizzati sulle macchine server (eventualmente remote). CORBA è stato progettato per un sistema eterogeneo costituito da diverse piattaforme hardware e diversi sistemi operativi, programmato con linguaggi diversi. Per consentire ad un client su una piattaforma di chiamare un server su una diversa piattaforma, sono interposte delle **ORB** (Object Request Brokers) tra il client e il server, che permettono loro di comunicare. Le ORB in CORBA svolgono un ruolo importante, fornendo anche il nome al sistema.

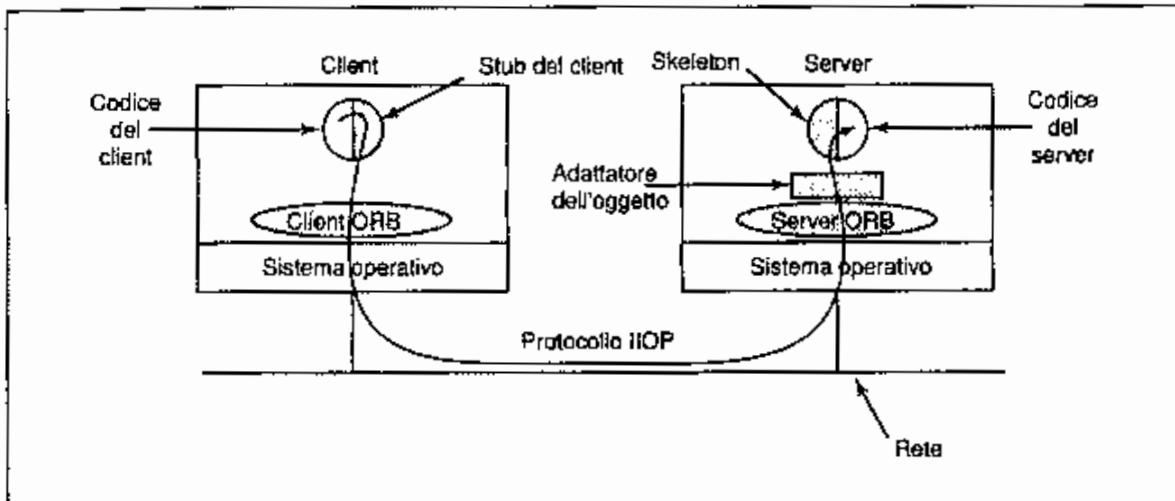
Ciascun oggetto CORBA è specificato tramite un'interfaccia, definita in un linguaggio chiamato **IDL** (Interface Definition Language), che determina quali metodi sia o esportati e il tipo dei loro parametri. È possibile compilare le specifiche IDL in una procedura stub del client, e memorizzarle in una libreria. Se un processo client sa in anticipo che avrà bisogno di accedere ad un determinato oggetto, viene collegato con il codice stub dell'oggetto client. È possibile compilare le specifiche IDL anche in una procedura **skeleton**, utilizzata sul versante del server. Se non si sa in anticipo quali oggetti CORBA occorrono ad un processo, è possibile una chiamata dinamica, ma questo esula dai confini della nostra trattazione.

Quando si crea un oggetto CORBA, si genera anche un riferimento ad esso, restituito al processo che ha creato l'oggetto stesso; per mezzo del riferimento, il processo identifica l'oggetto per le successive chiamate dei suoi metodi. Il riferimento può passare ad altri processi, o essere memorizzato in una directory di oggetti.

Per chiamare un metodo di un oggetto, un processo client deve prima acquisire un riferimento all'oggetto stesso, direttamente dal processo che lo ha creato, oppure tramite ricerca per nome o per funzione in una qualche sorta di directory. Una volta disponibile il riferimento all'oggetto, il processo client impacchetta i parametri delle chiamate al metodo in una struttura adatta, e quindi contatta il client ORB. A sua volta, quest'ultimo invia un messaggio al server ORB, che chiama effettivamente il metodo dell'oggetto: tutto questo meccanismo è simile ad una RPC.

Le ORB nascondono al codice del client e del server tutti i dettagli di basso livello, inerenti alla distribuzione e alla comunicazione. In particolare, nascondono al client la posizione del server, se il server abbia un programma binario o uno script, il tipo dell'hardware e il sistema operativo del server, se l'oggetto sia correntemente attivo, e come comunicano le due ORB (ad esempio TCP/IP, RPC, memoria condivisa eccetera).

Nella prima versione di CORBA, il protocollo fra il client e il server ORB non era specificato; di conseguenza ogni società produttrice di software ORB utilizzava un protocollo diverso, che non comunicava con gli altri. Nella versione 2.0, il protocollo è stato specificato: per le comunicazioni in Internet, il protocollo è detto **IOP** (Internet InterORB Protocol).



**Figura 8.38** Gli elementi principali di un sistema distribuito basato su CORBA; le parti CORBA sono colorate in grigio.

Per rendere possibile con i sistemi CORBA l'utilizzo di oggetti scritti per altri sistemi, si può munire ciascun oggetto di un adattatore. Questo è un involucro che gestisce compiti routinari, come registrare l'oggetto, generare riferimenti ad esso, ed attivarlo se chiamato quando non è attivo. L'organizzazione di queste componenti CORBA, è mostrata in Figura 8.38.

Un serio problema con CORBA è che ogni oggetto è localizzato su di un solo server, con conseguenti prestazioni assai ridotte, per oggetti molto utilizzati dalle macchine client sparse ovunque. In pratica, funziona in modo accettabile in sistemi di piccole dimensioni, ad esempio per connettere processi su un computer, su una LAN o nell'ambito di un'azienda.

## Globe

Come esempio di sistema distribuito basato sugli oggetti, appositamente progettato per estendersi a miliardi di utenti e a trilioni di oggetti, prendiamo in esame **Globe** (Van Steen et al., 1999a; Van Steen et al., 1999b). Due sono le idee chiave per i sistemi scalabili di grandi dimensioni: la prima è avere oggetti replicati; se esiste una sola copia di un oggetto cui vogliono accedere milioni di utenti sparsi nel mondo, l'oggetto sarà schiacciato dal peso delle richieste. Si pensi ad un oggetto che gestisca prezzi di merci o classifiche sportive: replicandolo, potremo distribuire il carico sulle copie.

La seconda idea è la flessibilità: in un sistema mondiale con un miliardo di utenti, non è possibile metterli d'accordo tutti su un linguaggio di programmazione, una strategia di duplicazione, un modello di sicurezza, o qualsiasi altro elemento. Il sistema deve permettere a utenti e oggetti diversi di comportarsi in modi diversi, fornendo nello stesso tempo un modello globale coerente. Questo è il compito svolto da **Globe**.

**Globe** come DSM, è insolito perché basato sul modello della memoria condivisa distribuita, applicata però ad un contesto planetario. In teoria, utilizzare in un sistema mondiale una normale DSM basata sulla paginazione, dovrebbe funzionare, ma le prestazioni sarebbero infime. Pertanto, **Globe** sceglie un diverso approccio: l'idea di fondo è che il mondo è pieno di oggetti, ciascuno dei quali contiene uno stato interno (nascosto) e metodi per accedere in modi controllati allo stato interno. Il segreto per rendere scalabile a livello mondiale la memoria condivisa consiste nel proibire LOAD e STORE dirette allo sta-

to interno di un oggetto, permettendo tutti gli accessi solo attraverso i metodi. Poiché un oggetto Globe può essere attivamente condiviso da molti processi nello stesso momento, è anche chiamato **oggetto condiviso distribuito**. La Figura 8.22 (c) mostra il posizionamento di sistemi come Globe.

Consideriamo ora come siano implementate la scalabilità e la flessibilità: ogni oggetto Globe ha un oggetto classe che contiene il codice effettivo per i suoi metodi. Ciascun oggetto dispone anche di una (o più) interfacce, ciascuna delle quali contiene coppie (puntatore al metodo, puntatore allo stato). Quindi, data un'interfaccia dell'oggetto, che è una tabella di puntatori presente in memoria a tempo di esecuzione, un processo può chiamare l'ennesimo metodo dell'oggetto, effettuando una chiamata alla procedura puntata dall'ennesima coppia nella tabella di interfaccia, e passando il corrispondente puntatore allo stato come parametro. Il puntatore allo stato è necessario, perché, se ci sono ad esempio due oggetti della classe mailbox in memoria, ciascuno ha la sua interfaccia con puntatori ai metodi condivisi, ma puntatori allo stato privati, come vediamo nella Figura 8.39. In quest'esempio il processo ha due caselle della posta aperte, ciascuna delle quali condivide il codice per i quattro metodi della classe mailbox, e ha il suo stato privato (i messaggi memorizzati nell'istanza della classe mailbox). Ad esempio, un oggetto mailbox potrebbe essere usato per la posta di lavoro, l'altro per quella personale.

Progettare le interfacce come tabelle in memoria a tempo di esecuzione, significa che gli oggetti non sono limitati ad un linguaggio particolare; questa decisione è dovuta al fatto che in un sistema planetario molte persone diverse sceglieranno molti linguaggi diversi. Un metodo di un oggetto può essere scritto in C, C++, Java o anche linguaggio assembly, se il possessore dell'oggetto lo desidera. Le interfacce proteggono il processo da ciò che sta dietro ai puntatori e ai metodi: questa architettura "mescola-e-collega" è più flessibile di una basata su un singolo linguaggio presente in alcuni sistemi (ad esempio, solo Java o solo C++).

Per utilizzare un oggetto Globe, un processo, dopo averlo cercato e aver trovato almeno un indirizzo di contatto (un indirizzo IP e una porta), deve collegarsi ad esso. Un controllo di sicurezza è effettuato al momento del collegamento, e se il processo è autorizza-

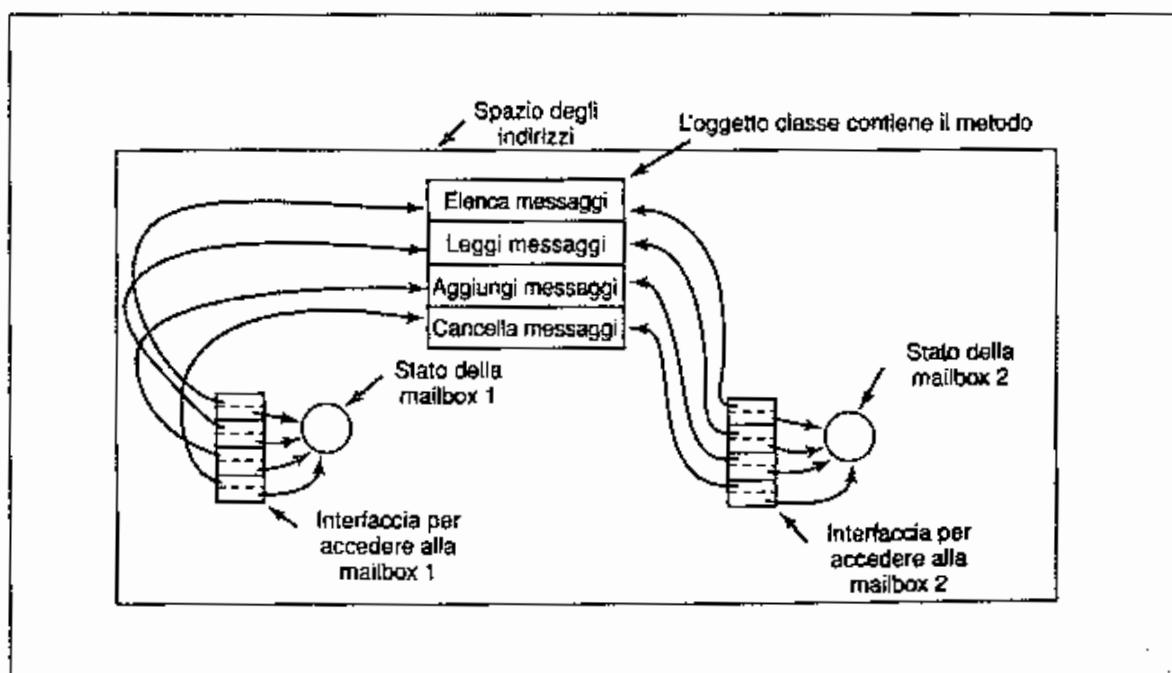
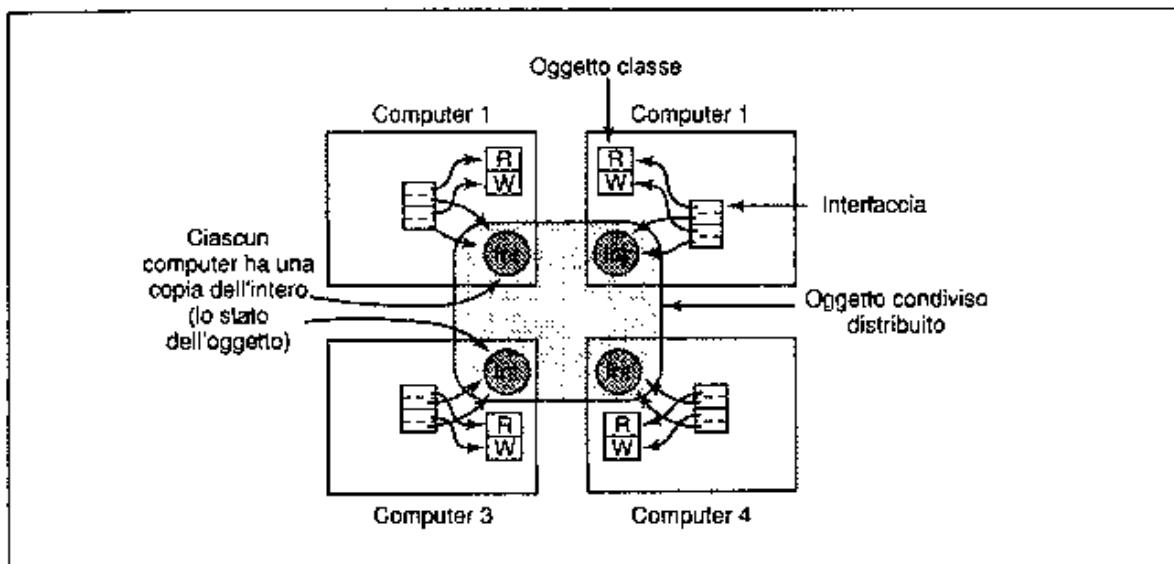


Figura 8.39 La struttura di un oggetto Globe.



**Figura 8.40** Un oggetto condiviso distribuito può avere il proprio stato copiato su diversi computer contemporaneamente.

to a collegarsi all'oggetto, l'oggetto della classe dell'oggetto (cioè il suo codice), è caricato nello spazio degli indirizzi del chiamante, una copia del suo stato viene istanziata, e viene restituito un puntatore alla sua interfaccia (standard). Utilizzando il puntatore dell'interfaccia, il processo può ora chiamare i metodi su questa istanza dell'oggetto; a seconda dell'oggetto, lo stato può essere quello di default, o una copia dello stato corrente, presa da una delle altre copie.

Immaginiamo l'oggetto più semplice possibile, con un intero come stato e due metodi, *read* e *write*, che operano sull'intero. Se processi diversi, in paesi diversi, sono simultaneamente collegati all'oggetto, tutti hanno una tabella di interfaccia che punta all'oggetto della classe contenente i due metodi (che sono caricati al momento del collegamento) come vediamo nella Figura 8.40. Ciascun processo (potenzialmente) ha anche una copia dell'intero corrispondente allo stato. Un metodo *read* può essere chiamato localmente, ma le *write* sono più complesse. Se l'oggetto vuole mantenere una consistenza sequenziale, deve fornire un meccanismo per farlo.

Un meccanismo consiste nell'avere un processo chiamato **sequenziatore**, il cui compito è fornire numeri consecutivi in sequenza, quando richiesti. Per effettuare una *write*, il metodo corrispondente dovrebbe prima acquisire un numero della sequenza e poi spedire a tutti gli altri processi collegati all'oggetto, un messaggio contenente numero della sequenza, nome dell'operazione e i parametri. Se due processi hanno chiamato simultaneamente una *write*, dovrebbero avere numeri di sequenza diversi; tutti i processi devono applicare i metodi in ordine di numero di sequenza, non nell'ordine d'arrivo dei messaggi. Se un processo acquisisce il numero di sequenza 26, ed il precedente era 24, deve aspettare il 25 prima di applicare il 26; se il 25 non compare entro un certo lasso di tempo, il processo deve attivarsi per localizzarlo e contattarlo. Questo schema garantisce che tutte le *write* siano fatte nello stesso ordine in tutte le copie dell'oggetto, assicurando la consistenza sequenziale.

Questa tecnica funziona abbastanza bene, ma non tutti gli oggetti hanno bisogno di consistenza sequenziale. Prendiamo ad esempio un oggetto che gestisca i prezzi di una partita di merci; se l'operatore aggiorna il prezzo per la partita 1, contemporaneamente ad un altro operatore che aggiorna la partita 2, non è essenziale che tutte le copie del-

l'oggetto applichino i due aggiornamenti nell'ordine in cui si sono verificati, perché sono indipendenti. Può bastare che tutti i processi applichino gli aggiornamenti degli operatori nell'ordine in cui sono stato spediti, ma l'obiettivo si può raggiungere includendo un numero di sequenza generato dal processo mittente, ed in questo caso non è necessario un sequenziatore esteso a tutti gli oggetti.

Lo schema precedente, cioè un oggetto replicato con tutte le copie uguali ed in grado di eseguire aggiornamenti, dopo aver ottenuto un numero di sequenza, è solo uno dei molti protocolli di replicazione. Un altro schema ha una copia master di ciascun oggetto, più un certo numero di copie slave; tutti gli aggiornamenti sono spediti alla copia master, che quindi applica gli aggiornamenti e spedisce il nuovo stato a tutte le copie slave.

Una terza strategia di replicazione degli oggetti consiste nell'avere una sola copia che memorizzi lo stato dell'oggetto, mentre tutte le altre sono considerate rappresentanti senza stato. Quando si effettuano una *read* o una *write* su una di queste copie (ad esempio, una macchina client), la richiesta viene inoltrata alla copia che mantiene lo stato, ed ivi eseguita.

La forza di Globe è che ciascun oggetto può avere la sua strategia di replicazione. Alcuni oggetti utilizzano la replicazione attiva, mentre altri usano la replicazione master-slave, o altre strategie eventualmente necessarie per l'oggetto. Inoltre, ciascun oggetto può avere la sua strategia riguardo alla consistenza, alla creazione e rimozione di copie, alla sicurezza, eccetera, e questo è possibile perché tutte le strategie sono gestite all'interno dell'oggetto: gli utenti dell'oggetto non se ne devono occupare, e neppure gli amministratori di sistema. Questo approccio è diverso da quello di CORBA, che non nasconde nessuna di queste strategie dentro gli oggetti, rendendo in tal modo difficile avere mille oggetti con mille strategie diverse.

Un oggetto Globe può essere implementato come nella Figura 8.41, che illustra i sotto-oggetti di cui è composto un oggetto Globe. L'oggetto di controllo accetta le chiamate ai metodi di ingresso e utilizza gli altri sotto-oggetti per eseguirle; il sotto-oggetto della semantica esegue effettivamente il lavoro richiesto dall'interfaccia dell'oggetto, ed è la

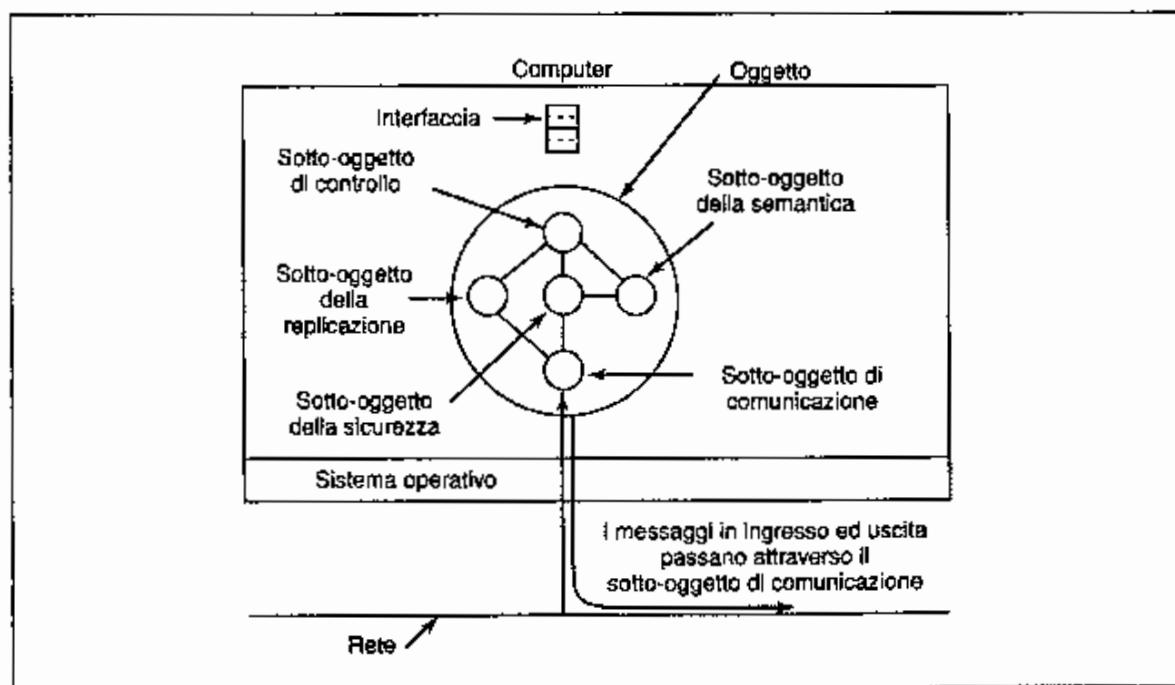


Figura 8.4 1 Struttura di un oggetto Globe.

sola parte del codice dell'oggetto che dev'essere scritta dal programmatore, tutto il resto si può ricavare da librerie standard, salvo che il programmatore non voglia implementare una nuova strategia di norma non disponibile. Il compito del sotto-oggetto di replicazione consiste nel gestire la replicazione, può essere rimpiazzato per passare da una replicazione attiva ad una master-slave, o ad altre strategie di replicazione, senza che il resto dell'oggetto ne sia coinvolto. Allo stesso modo, il sotto-oggetto sicurezza può essere rimpiazzato per implementare una nuova strategia di sicurezza (ad esempio, per passare da ACL alle capability) e il sotto-oggetto di comunicazione per cambiare i protocolli di rete (ad esempio da IP v4 a IP v6) senza influenzare il resto dell'oggetto.

Per vedere come interagiscono questi sotto-oggetti, osserviamo cosa accade quando è chiamato uno dei metodi. Il codice puntato dall'interfaccia si trova nel sotto-oggetto di controllo, che richiede poi al sotto-oggetto di replicazione di svolgere il suo compito; se l'oggetto è replicato in modo attivo, viene prima acquisito un numero di sequenza. Quindi il sotto-oggetto di replicazione dice a tutte le copie (compresa la sua) di fare effettivamente il lavoro, chiamando il loro oggetto di semantica; se l'oggetto addotta una strategia master-slave, e il metodo viene chiamato da uno slave, viene spedito un messaggio al master, e così via. In momenti opportuni, si effettuano controlli di sicurezza dall'oggetto relativo (per verificare se la chiamata è permessa, se i dati uscenti devono essere crittati ecc).

Un elemento chiave di Globe è il server di localizzazione, che permette il ritrovamento degli oggetti ovunque nel mondo. Tale servizio è costruito come un albero, con le registrazioni degli oggetti mantenute solo nel nodo dove avviene la registrazione. I puntatori a questo nodo sono propagati verso la cima dell'albero in modo che sia sempre possibile trovare la registrazione. La località, il partizionamento dei nodi dell'albero, i meccanismi di cache e altre tecniche si utilizzano per rendere lo schema praticabile, anche per oggetti mobili (Ballintijn et al., 2000; Van Steen et al., 1998a; Van Steen et al., 1998b).

### 8.3.6 Middleware basato sulla coordinazione

Il nostro ultimo paradigma per un sistema distribuito è chiamato **middleware basato sulla coordinazione**. Inizieremo con il sistema Linda, un progetto di ricerca universitario che ha dato l'avvio a questi studi, quindi vedremo due esempi commerciali, molto dipendenti da Linda, publish/subscribe e Jini.

#### Linda

Linda è un nuovo sistema per la comunicazione e la sincronizzazione sviluppato a Yale da David Gelernter e dal suo studente Nick Carriero (Carriero e Gelernter 1986, 1989 Gelernter 1985). In Linda processi indipendenti comunicano attraverso uno spazio delle tuple astratto. Lo spazio delle tuple è globale all'intero sistema, e i processi su una macchina possono inserire o rimuovere tuple dal loro spazio senza preoccuparsi come o dove esse siano memorizzate. All'utente, lo spazio delle tuple appare come una grande memoria condivisa globale, come abbiamo visto in precedenza in diverse forme [Figura 8.22 (c)].

Una tupla è come una struttura in C o un record in Pascal, composta da uno o più campi, ciascuno dei quali ha un valore di un certo tipo supportato dal linguaggio base (Linda è implementata aggiungendo una libreria ad un linguaggio esistente, come ad esempio il C). Per il C-Linda, i tipi dei campi sono interi, long, e numeri in virgola mobile, e tipi composti come gli array (incluse le stringhe) e le strutture (ma non altre tuple). Diversamente dagli oggetti, le tuple sono dati puri, e non hanno metodi associati. La Figura 8.42 mostra tre tuple di esempio.

```
[“abc”, 2, 5)
[“matrix-1”, 1, 6, 3.14)
(“famiglia”, “è-sorella”, “Stefania”, “Roberta”)
```

**Figura 8.42** Tre tuple di Linda.

Sono fornite quattro operazioni sulle tuple. La prima, *out*, mette una tupla nello spazio delle tuple. Ad esempio,

```
out(“abc”, 2, 5);
```

mette la tupla ( “abc”, 2, 5) nello spazio delle tuple. I campi di *out* sono normalmente costanti, variabili o espressioni, come in

```
out(“matrix - 1”, i, j, 3.14);
```

che manda in uscita una tupla con quattro campi, il secondo e il terzo dei quali sono determinati dai valori correnti delle variabili *i* e *j*.

Le tuple sono recuperate dallo spazio delle tuple dalla primitiva *in*. Sono ricercate per contenuto, piuttosto che per nome o indirizzo; i campi di *in* possono essere espressioni o parametri formali. Si consideri, per esempio,

```
in(“abc”, 2, ?i);
```

Questa operazione cerca nello spazio delle tuple, una tupla che consista della stringa ‘abc’, dell’intero 2, e un terzo campo contenente un intero qualsiasi (supponendo che *i* sia un intero). Se viene trovata, la tupla è rimossa dallo spazio, assegnando alla variabile *i* il valore del terzo campo. Le operazioni di ricerca della corrispondenza e di rimozione sono atomiche, cosicché se due processi eseguono simultaneamente la stessa operazione *in*, ne avverrà una sola, salvo che non siano presenti due o più tuple corrispondenti. Lo spazio delle tuple può anche contenere più copie della stessa tupla.

L’algoritmo di ricerca delle corrispondenze utilizzato da *in* è semplice. I campi della primitiva *in*, chiamati template, sono (concettualmente) confrontati con i campi corrispondenti di ogni tupla nello spazio delle tuple. Una corrispondenza si verifica se sono soddisfatte le seguenti tre condizioni:

1. Il template e la tupla hanno lo stesso numero di campi.
2. I tipi dei campi corrispondenti sono uguali.
3. Ciascuna costante o variabile nel template corrisponde al campo della tupla

I parametri formali, indicati da un punto interrogativo seguito da un nome di variabile o da un tipo, non partecipano alla ricerca (eccetto che per il controllo di tipo), benché a quelli contenenti un nome di variabile venga assegnato un valore dopo il verificarsi di una corrispondenza.

Se non ci sono tuple corrispondenti, il processo corrispondente viene sospeso, finché un altro processo inserisce la tupla di cui ha bisogno e, a quel punto, il chiamante è risvegliato automaticamente, e gli viene assegnata la nuova tupla. Il fatto che i processi si bloccino e si sblocchino automaticamente, significa che se un processo sta per mandare in

uscita una tupla, e un altro sta per riceverla, non importa chi arrivi primo. La sola differenza è che se la *in* viene effettuata prima della *out*, ci sarà un piccolo ritardo prima che la tupla sia disponibile per la rimozione.

Il fatto che i processi si blocchino quando non è presente una tupla di cui hanno bisogno, può essere utilizzato in molti modi, ad esempio, per implementare i semafori. Per creare o effettuare un'operazione di *up* sul semaforo *S*, un processo può eseguire

```
Out("semaforo S");
```

Per effettuare una *down* il processo può eseguire

```
In("semaforo S");
```

Lo stato di un semaforo *S* è determinato dal numero di ("semaforo *S*") nello spazio delle tuple. Se non ne esiste nessuno, ogni tentativo di ottenerne uno bloccherà il processo finché un altro processo non gliene fornirà uno.

Oltre a *out* e *in*, Linda ha anche una primitiva *read*, uguale a *in*, a parte che non rimuove la tupla dallo spazio delle tuple. Esiste anche una primitiva *eval*, i cui parametri sono valutati in parallelo e la tupla risultante viene messa nello spazio delle tuple. Questo meccanismo può essere utilizzato per effettuare un calcolo arbitrario, e permette di creare processi paralleli in Linda.

## Publish/subscribe

Il prossimo esempio basato sulla coordinazione è ispirato a Linda, ed è chiamato **publish/subscribe** (Oki et al., 1993). Si compone di un certo numero di processi connessi da una rete broadcast; ciascuno dei quali può essere produttore o consumatore di informazioni, oppure entrambi.

Quando un produttore ha delle nuove informazioni (ad esempio, nuovi prezzi di una partita di merci), trasmette a tutti nella rete l'informazione come una tupla; questa azione è detta **pubblicazione (publish)**. Ciascuna tupla contiene una linea con soggetto gerarchico, con campi multipli separati da virgolette. I processi interessati a determinate informazioni possono **sottoscrivere (subscribe)** determinati soggetti, includendo anche

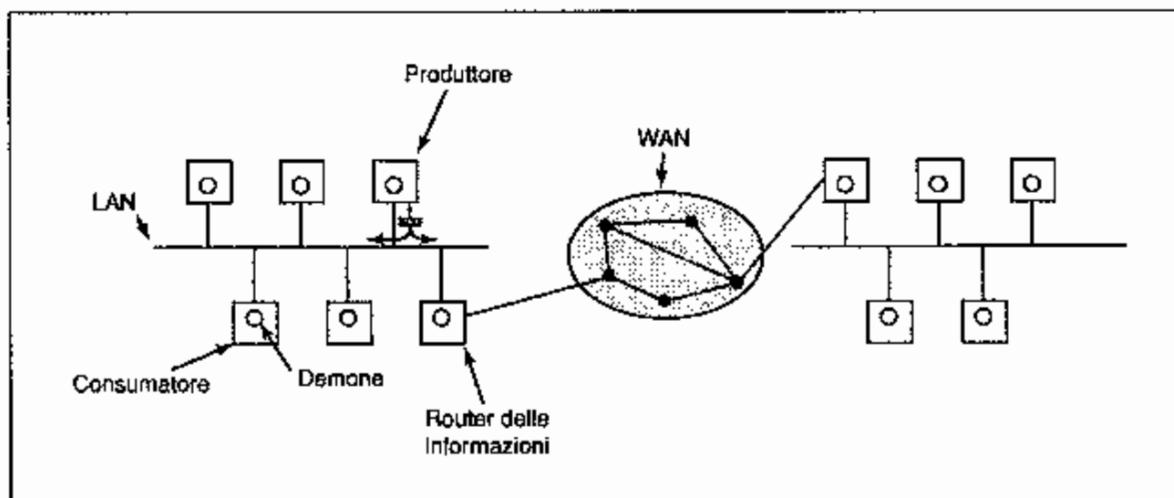


Figura 8.43 L'architettura publish/subscribe.

caratteri jolly nella linea soggetto. Una sottoscrizione si effettua richiedendo quali soggetti cercare ad un processo demone delle tuple in esecuzione sulla stessa macchina, che controlla le tuple pubblicate.

Publish/subscribe è implementato come vediamo in Figura 8.43. Quando un processo deve pubblicare una tupla, la trasmette sulla LAN locale; il demone delle tuple di ciascuna macchina copia tutte le tuple trasmesse nella propria LAN, quindi controlla la linea soggetto per vedere quali processi siano interessati alla tupla, inoltrando una copia a ciascun processo interessato. È possibile anche ritrasmettere le tuple su una rete geografica o su Internet, tramite una macchina su ciascuna LAN che agisca come router di informazioni, raccogliendo cioè tutte le tuple pubblicate e inoltrandole ad altre LAN, perché vengano ritrasmesse. Questa operazione si può anche compiere intelligentemente, inoltrando una tupla ad una LAN remota, solo se quella LAN ha almeno un sottoscrittore che voglia la tupla. Ciò richiede che i router delle informazioni si scambino informazioni sui sottoscrittori.

Si possono implementare diversi tipi di semantica, come la spedizione affidabile e la spedizione garantita, anche in presenza di crash; in questo caso, occorre memorizzare le vecchie tuple, nel caso siano utili in seguito. Un modo per memorizzarle consiste nel collegare un database al sistema, affinché sottoscriva tutte le tuple, impacchettando il database in un adattatore, in modo che possa funzionare con il modello publish/subscribe. Quando arriva una tupla, l'adattatore la cattura e la mette nel database.

Il modello publish/subscribe separa pienamente i produttori dai consumatori, come fa Linda, ma talvolta è utile conoscerne l'identità. Questa informazione si può acquisire pubblicando una tupla che fondamentalmente chiede: "chi è interessato a *x*?" le risposte ritornano in forma di tuple che dicono: "io sono interessata a *x*".

## Jini

Per oltre cinquant'anni la computazione è stata CPU-centrica, con un computer concepito come un dispositivo indipendente, costituito da una CPU, dalla memoria primaria e spesso da un dispositivo di memoria di massa, come un disco. Jini, (n.d.t.: si pronuncia come il termine genio, in inglese *genie*), della Sun Microsystems, rappresenta un tentativo di cambiare il modello tradizionale per uno definibile rete-centrico (Waldo 1999).

Il mondo Jini si compone di un gran numero di dispositivi Jini autocontenuti, ciascuno dei quali offre uno o più servizi agli altri; un dispositivo Jini può essere inserito in una rete, ed istantaneamente inizia ad offrire e utilizzare servizi, senza richiedere complesse procedure di installazione. Si noti che i dispositivi sono inseriti in una rete, non in un computer come avviene di norma; un dispositivo Jini può essere un computer tradizionale, una stampante, un palmare, un cellulare, un televisore, uno stereo o qualunque altro dispositivo con una CPU, un poco di memoria e una connessione di rete (eventualmente wireless, senza cavi). Un sistema Jini è una federazione lasca di dispositivi Jini, che possono entrare e uscire a piacere, senza amministrazione centrale.

Quando un dispositivo Jini vuole unirsi alla federazione, trasmette un pacchetto alla LAN locale, o nella cella wireless locale, chiedendo se è presente un servizio di ricerca. Per trovare un servizio di ricerca si usa il protocollo di scoperta, uno dei pochi cablati in hardware di Jini. (In alternativa, il nuovo dispositivo Jini può attendere, fino all'arrivo di uno degli annunci periodici di un servizio di ricerca, ma non ci occuperemo di questo meccanismo).

Quando un servizio di ricerca vede che un nuovo dispositivo vuole registrarsi, risponde con una porzione di codice che può effettuare la registrazione. Poiché Jini si basa su

Java, il codice spedito è in JVM (il linguaggio della Java Virtual Machine), che tutti i dispositivi Jini devono essere in grado di eseguire, di solito in modo interpretato. Il nuovo dispositivo esegue ora il codice, che contatta il servizio di ricerca e si registra presso il servizio, per un periodo di tempo determinato. Prima della scadenza, il dispositivo, se lo desidera, può registrarsi nuovamente; ciò significa che un dispositivo Jini può abbandonare il sistema semplicemente spegnendosi, e la sua precedente esistenza sarà presto dimenticata, senza bisogno di amministrazione centrale. La registrazione per un intervallo di tempo fissato è detta **acquisire un lease**.

Si noti che, poiché il codice per registrare il dispositivo è scaricato sul dispositivo stesso, può essere cambiato man mano che il sistema si evolve, senza ripercussioni sull'hardware e il software del dispositivo: esso, infatti, ignora quale sia il protocollo di registrazione. Una parte del processo di registrazione sul quale il dispositivo è informato, consiste nel fornire alcuni attributi e una parte di codice, che gli altri dispositivi useranno in seguito per accedere al dispositivo stesso.

Un dispositivo o un utente in cerca di un particolare servizio, può chiedere al servizio di ricerca se ne conosca uno; la richiesta può coinvolgere alcuni degli attributi utilizzati dal dispositivo durante la registrazione. Se la richiesta ha successo, il codice che il dispositivo ha fornito durante la registrazione è mandato al richiedente, ed eseguito per contattare il dispositivo. Quindi un dispositivo, o un utente, possono dialogare con un altro dispositivo senza sapere dove esso si trovi e quale protocollo utilizzi.

Il client e i servizi Jini (dispositivi hardware o software) comunicano e si sincronizzano usando gli **JavaSpace**, modellati sullo spazio delle tuple di Linda, con alcune significative differenze. Ogni JavaSpace si compone di un certo numero di entry fortemente tipate (la differenza con le tuple di Linda è proprio che sono fortemente tipate, mentre le tuple di Linda non lo sono). Ciascuna entry si compone di un certo numero di campi, ciascuno dei quali ha un tipo base di Java, ad esempio una entry di tipo "impiegato" potrebbe essere fatta da una stringa (per il nome), un intero (per il settore), un secondo intero (per il telefono) e un valore booleano (per segnalare se lavora a tempo pieno).

Nello JavaSpace sono definiti quattro metodi (benché due presentino una variante):

1. **Write:** mette una nuova entry nello JavaSpace.
2. **Read:** copia una entry che corrisponde ad un template dallo JavaSpace.
3. **Take:** copia e rimuove una entry che corrisponde ad un template.
4. **Notify:** notifica al chiamante quando viene scritta una entry corrispondente.

Il metodo *write* fornisce l'entry e specifica l'intervallo di tempo della lease, cioè quando dovrebbe essere eliminata (al contrario, le tuple di Linda rimangono finché vengono esplicitamente rimosse). Uno JavaSpace può contenere più volte la stessa entry, quindi non è un insieme matematico (come in Linda).

I metodi *read* e *take* forniscono un template per la entry che si sta cercando. Ciascun campo del template può contenere un valore specifico, che deve corrispondere, oppure un carattere jolly che corrisponde a tutti i valori del tipo appropriato; se si trova una corrispondenza essa è restituita, e nel caso di *take*, è anche rimossa dallo JavaSpace. Ciascuno di questi metodi JavaSpace, ha due varianti che differiscono nel caso non esistano entry corrispondenti. Una variante ritorna immediatamente con un'indicazione di fallimento, l'altra aspetta per un certo intervallo di tempo (dato come parametro).

Il metodo *notify* serve per manifestare un interesse per un particolare template. Introducendo in seguito una entry corrispondente, viene chiamato il metodo *notify* del chiamante.

A differenza dello spazio delle tuple di Linda, lo JavaSpace supporta le transazioni

atomiche. Utilizzandole, si possono raggruppare metodi diversi, che verranno eseguiti tutti insieme, oppure nessuno. Durante la transazione, i cambiamenti fatti allo JavaSpace non sono visibili al di fuori della transazione stessa; solo quando la transazione è confermata divengono visibili agli altri chiamanti.

Si può utilizzare JavaSpace per sincronizzare processi comunicanti. Ad esempio, in una situazione produttore-consumatore, il primo pone degli elementi nello JavaSpace appena li produce, il secondo li rimuove con *take*, bloccandosi se non esiste alcun elemento disponibile. Lo JavaSpace garantisce che ciascuno dei metodi sia eseguito in modo atomico, cosicché non si rischia che un processo tenti di leggere una entry inserita solo a metà.

## 8.4 Ricerca sui sistemi a processori multipli

In questo capitolo abbiamo considerato tre tipi di sistemi a processori multipli: i multiprocessori, i multicomputer ed i sistemi distribuiti. Diamo anche un breve sguardo alla ricerca in queste tre aree. Gran parte della ricerca sui multiprocessori è relativa all'hardware, in particolare a come costruire la memoria condivisa ed a come mantenerla coerente. In ogni modo, la ricerca si è anche occupata dell'utilizzo dei monitor della macchina virtuale con i multiprocessori (Bugnion et al., 1997) e sulla gestione delle risorse nei multiprocessori (Govil et al., 1999). La schedulazione dei thread interessa anche l'algoritmo di schedulazione (Arora et al., 1998; Philbin et al., 1996), e la contesa per la coda di esecuzione (Dandamudi, 1997).

I multicomputer sono molto più facili da costruire dei multiprocessori; tutto ciò che ci vuole è un insieme di PC o workstation ed una rete ad alta velocità, e per questa ragione, sono un argomento di ricerca diffuso nelle università. Molti studi si occupano delle forme di memoria condivisa distribuita, talvolta basata sulla paginazione ma in altri casi realizzata interamente in software (Carter et al., 1995; Feeley et al., 1995; Johnson et al., 1995; Itzkovitz e Schuster, 1999; Scales e Gharachorloo, 1997; Stets et al., 1997). Un altro argomento di ricerca è l'ottimizzazione della comunicazione a livello utente (Von Eicken et al., 1995), così come il bilanciamento del carico (Harchol-Balter e Downey, 1996).

Sono stati inoltre pubblicati molti articoli sui sistemi distribuiti, per esempio sul middleware (Bernstein, 1996), sugli oggetti (Dogac et al., 1998), sui sistemi senza cavi (Liu et al., 1996), sugli agenti mobili (Chen et al., 2000), sugli ambienti di programmazione (Jo, 1999), sui multimedia distribuiti (Mourlas, 2000), sulla teoria (Buchs e Guelfi, 2000), e sui meccanismi di cache per il Web (Wolman et al., 1999), ed anche altri. Ci si è inoltre occupati dei file system distribuiti (Alexandrov et al., 1998; Hartman e Ousterhout, 1995; Thekkat et al., 1997) e dei file system mobili (Segarra e Andri, 1999).

## 8.5 Sommario

È possibile rendere più veloci e affidabili i sistemi di computer, utilizzando CPU multiple. Le tre principali organizzazioni di sistemi dotati di più CPU sono rappresentati dai multiprocessori, dai multicomputer e dai sistemi distribuiti. Ogni sistema ha le sue proprietà e i suoi problemi.

Un multiprocessore si compone di due o più CPU che condividono una RAM comune; le CPU possono essere interconnesse tramite un bus, una connessione crossbar, o una rete di connessione a più stadi. Sono possibili diverse configurazioni del sistema operativo: ogni CPU con il proprio sistema operativo; un sistema operativo master e gli altri con-

figurati come slave; multiprocessori simmetrici, in cui si ha una copia del sistema operativo eseguibile da qualsiasi CPU. In quest'ultimo caso sono necessari lock per ottenere la sincronizzazione; quando un lock non è disponibile, una CPU può effettuare lo spin o un cambiamento di contesto. Sono possibili diversi algoritmi di schedulazione, tra i quali il timesharing, la condivisione dello spazio e la schedulazione-gang.

I multicomputer hanno anch'essi due o più CPU, ma queste CPU hanno ciascuna la loro memoria privata; non condividono una memoria comune e quindi comunicano tramite lo scambio di messaggi. In molti casi la scheda dell'interfaccia di rete possiede la sua propria CPU, pertanto occorre organizzare attentamente la comunicazione fra la CPU principale e la CPU della scheda di interfaccia, al fine di evitare le corse critiche. La comunicazione a livello utente nei multicomputer viene spesso realizzata attraverso le chiamate di procedura remota, ma anche attraverso la memoria condivisa distribuita. In questo contesto, il bilanciamento del carico dei processi è problematico, ed esistono vari algoritmi per affrontarlo, tra i quali gli algoritmi iniziati dal mittente e dal ricevente, e quelli basati sull'offerta.

I sistemi distribuiti sono debolmente connessi; ciascuno dei loro nodi è un computer completo, con una dotazione completa di periferiche e il proprio sistema operativo; di frequente, questi sistemi si estendono su ampie aree geografiche. Spesso il middleware è sovrapposto al sistema operativo, per fornire uno strato uniforme per mezzo del quale le applicazioni possano interagire. Tra i diversi tipi, abbiamo il middleware basato sui documenti, quello basato sui file e quello basato sulla coordinazione. Alcuni esempi sono il World Wide Web, AFS, CORBA, Globe, Linda e Jinu.

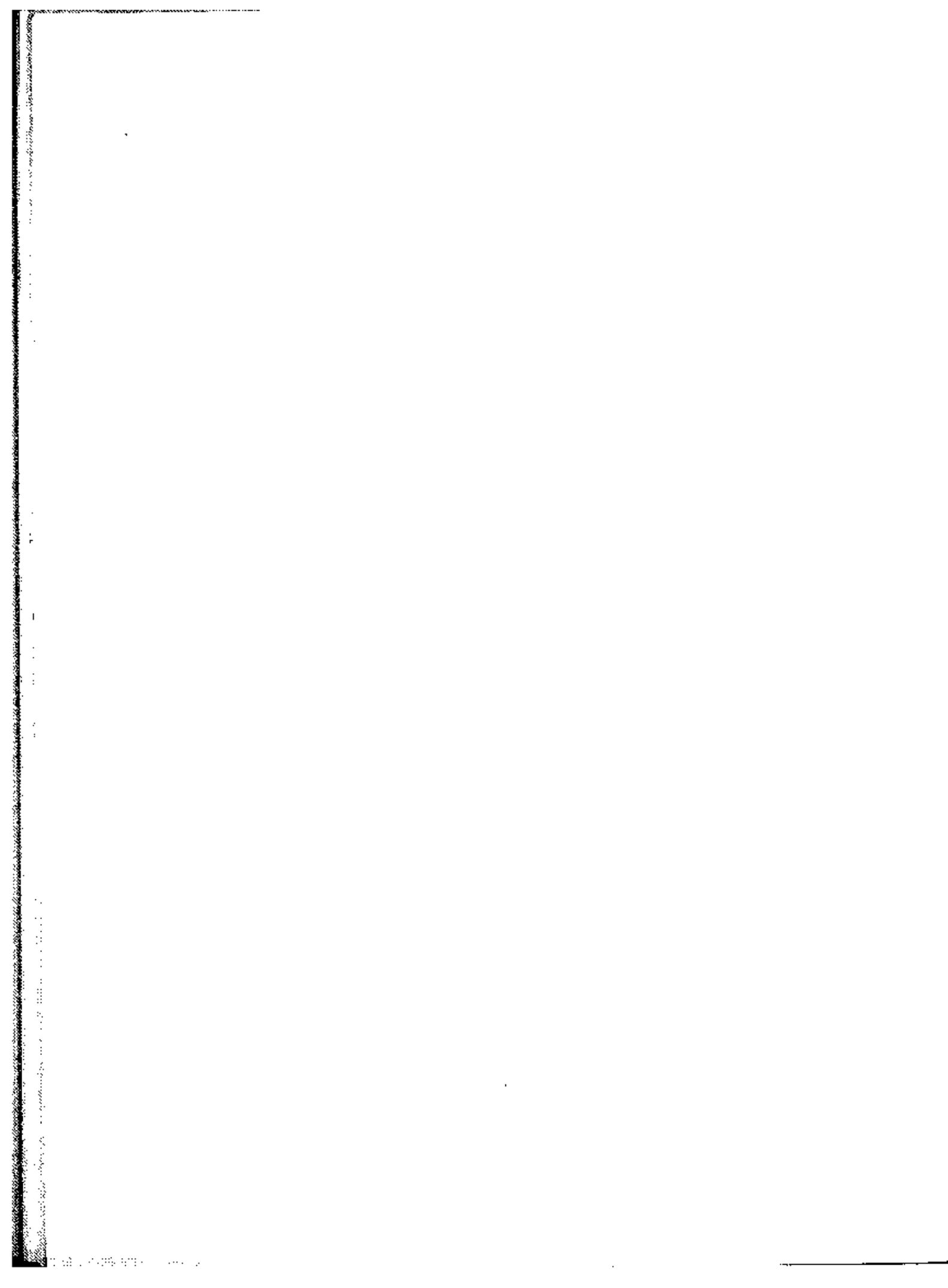
## PROBLEMI

1. Possono essere considerati sistemi distribuiti il newsgroup USENET o il progetto SETI@home (SETI@home utilizza svariati milioni di personal computer inattivi per analizzare i dati dei radiotelescopi per cercare forme di vita intelligente extraterrestre). In caso affermativo, come sono in relazione con le categorie descritte nella Figura 8.1?
2. Che cosa accade se due CPU in un multiprocessore tentano di accedere nello stesso istante alla stessa parola di memoria?
3. Se una CPU effettua una richiesta di memoria ogni istruzione, e il computer esegue a 200 MIPS, all'incirca quante CPU saranno necessarie per saturare un bus a 400 MHz? Si supponga che un riferimento in memoria richieda un ciclo di bus. Quindi, si ripeta l'esercizio per un sistema in cui si usi la cache, con un tasso di successo del 90%; infine, quale tasso di successo della cache sarebbe necessario per permettere a 32 CPU di condividere il bus senza sovraccaricarlo?
4. Si supponga che il cavo fra il connettore 2A e il connettore 3B nella rete omega della Figura 8.5 si rompa. Chi resta isolato da chi?
5. Come si effettua la gestione del segnale nel modello della Figura 8.7?
6. Quando una chiamata di sistema viene fatta nel modello della Figura 8.8, dev'essere immediatamente risolto un problema dopo la trap, che non si verifica nel modello della Figura 8.7. Qual è la natura di questo problema, e come potrebbe essere risolto?
7. Si riscriva il codice di *entra\_nella\_regione* della Figura 2.22, utilizzando una read pura per ridurre il trashing indotto dall'istruzione TSL.

8. In un sistema operativo SMP, sono realmente necessarie le regioni critiche nelle sezioni di codice per evitare le corse critiche, oppure tale compito può essere svolto bene anche da mutex sulle strutture dati?
9. Quando l'istruzione TSL è usata per la sincronizzazione dei multiprocessori, il blocco della cache che contiene il mutex fa la spola fra la CPU che mantiene il lock e quella che lo richiede, se entrambe utilizzano il blocco stesso. Per ridurre il traffico del bus, la CPU richiedente esegue una TSL ogni 50 cicli di bus, mentre la CPU che detiene il lock accede sempre al blocco della cache fra le istruzioni TSL. Se un blocco della cache si compone di 16 parole a 32 bit, ciascuna delle quali richiede un ciclo di bus per il trasferimento, e il bus ha una frequenza di 400 MHz, quale frazione della larghezza di banda del bus viene sprecata muovendo il blocco della cache avanti e indietro?
10. Nel testo è stato suggerito di usare un algoritmo di backoff esponenziale binario al posto di una TSL, per effettuare il poll di un lock. È stato anche suggerito di porre un ritardo massimo fra i cicli di poll. L'algoritmo funzionerà correttamente se non viene stabilito un ritardo massimo?
11. Si supponga che non sia disponibile l'istruzione TSL per sincronizzare un multiprocessore, e che sia invece disponibile un'altra istruzione, SWP, che scambia atomicamente i contenuti di un registro con una parola in memoria. Potrebbe essere utilizzata per la sincronizzazione di un multiprocessore? In caso affermativo, in che modo? In caso negativo, perché non funziona?
12. In questo problema, si deve calcolare quanto carico sul bus venga posto da uno spin lock. Si immagini che ciascuna istruzione eseguita da una CPU richieda 5 ns. Dopo che è stata completata un'istruzione, sono eseguiti i cicli di bus richiesti, ad esempio da una TSL. Ciascun ciclo di bus richiede 10 ns addizionali, prima e dopo il tempo di esecuzione dell'istruzione. Se un processo sta tentando di entrare in una regione critica, utilizzando un loop TSL, quale frazione della larghezza di banda del bus consuma? Si supponga che stia funzionando un normale meccanismo di cache, in modo che il prelievo di un'istruzione all'interno del loop non consumi cicli di bus.
13. Si è detto che la Figura 8.12 illustra un ambiente timesharing. Perché è mostrato solo un processo (A) nella parte (b)?
14. La schedulazione per affinità riduce le miss della cache. Riduce anche le miss TLB? Cosa si può dire riguardo ai fault di pagina?
15. Per ciascuna delle topologie della Figura 8.16, qual è il diametro della rete di interconnessione? Si contino allo stesso modo tutti i passi (host-router e router-router).
16. Si consideri la topologia a doppio toro della Figura 8.16 (d), ma espansa alla dimensione  $k \times k$ . Qual è il diametro della rete? Suggerimento: si considerino separatamente i casi  $k$  dispari e  $k$  pari.
17. La larghezza di banda di bisezione di una rete di interconnessione è spesso usata come misura della sua capacità, ed è calcolata rimuovendo il numero minimo di link che suddividono la rete in due unità di uguali dimensioni, e poi sommando la capacità dei link rimossi. Se ci sono molti modi di effettuare la suddivisione, quella con la minima larghezza di banda è la larghezza di banda di bisezione. Per una rete di interconnessione, costituita da un cubo  $8 \times 8 \times 8$ , qual è la larghezza di banda di bisezione, se ciascun link è un Gbps?

18. Si consideri un multicomputer in cui l'interfaccia di rete funzioni in modalità utente, in modo che siano necessarie solo tre operazioni di copia, dalla RAM sorgente alla RAM destinazione. Si supponga che spostare una parola di 32 bit da o verso la scheda di interfaccia di rete richieda 20 ns, e che la rete stessa operi a 1 Gbps. Quale sarebbe il ritardo per un pacchetto di 64 byte spedito dalla sorgente alla destinazione, se potessimo ignorare il tempo di copia? Quale sarebbe il ritardo, considerando il tempo di copia? Si consideri ora il caso in cui siano necessarie altre due copie, al kernel sul versante del mittente, e dal kernel, sul versante del ricevente. Qual è il ritardo in questo caso?
19. Si ripeta il precedente esercizio, per il caso delle tre e delle cinque copie, calcolando però la larghezza di banda invece del ritardo.
20. In cosa differisce l'implementazione di una send e di una receive fra un sistema multiprocessore a memoria condivisa e un multicomputer, e che effetti ha sulle prestazioni?
21. Quando si trasferiscono dati da una RAM all'interfaccia di rete, si può usare il blocco di una pagina. Si supponga però che le chiamate di sistema per il blocco e lo sblocco di una pagina richiedano ciascuna 1  $\mu$ s. La copia richiede 5 byte/ns, utilizzando il DMA, ma 20 ns/byte utilizzando l'I/O programmato. Quanto dev'essere grande un pacchetto perché sia vantaggioso il blocco della pagina e l'utilizzo del DMA?
22. Quando una procedura è raccolta da una macchina e sistemata su un'altra, per essere chiamata tramite RPC, possono verificarsi alcuni problemi. Nel testo ne abbiamo rilevato quattro: puntatori, dimensioni sconosciute degli array, tipi sconosciuti dei parametri, e variabili globali. Un aspetto non discusso, è cosa accade se la procedura remota esegue una chiamata di sistema. Quali problemi potrebbero sorgere? Come si potrebbero gestire?
23. In un sistema DSM, quando avviene un fault di pagina, occorre localizzare la pagina di cui si ha bisogno. Si indichino due possibili modi per trovare la pagina.
24. Si consideri l'allocazione dei processori della Figura 8.25. Si supponga che il processo H sia spostato dal nodo 2 al nodo 3. Qual è ora il peso totale del traffico esterno?
25. Alcuni multicomputer permettono ai processi in esecuzione di migrare da un nodo ad un altro. È sufficiente fermare un processo, congelare la sua immagine di memoria e spedirlo a un nodo diverso? Elenca due problemi non banali che devono essere risolti per permettere questa operazione.
26. Perché c'è un limite alla lunghezza del cavo in una rete Ethernet?
27. Nella Figura 8.28 il terzo e il quarto strato sono etichettati middleware e applicazione su tutte e quattro le macchine. In che senso sono uguali nelle diverse piattaforme, e in che senso sono differenti?
28. La Figura 8.31 elenca sei tipi diversi di servizi. Per ciascuna delle seguenti applicazioni, qual è il tipo di servizio più appropriato?
  - (a) Video su richiesta in Internet.
  - (b) Scaricare una pagina Web.
29. I nomi DNS hanno una struttura gerarchica, *cs.uni.edu* o *sale.general-widget.com*. Si potrebbe gestire il database DNS attraverso un database centralizzato, ma non si fa, perché riceverebbe troppe richieste ogni secondo. Si faccia una proposta su come gestire, in pratica, il database.

30. Nella discussione sull'elaborazione degli URL da parte di un browser, s'era detto che le connessioni sono fatte alla porta 80. Perché?
31. Gli URL usati nel Web sono un esempio di trasparenza di localizzazione, o no? Si motivi la risposta.
32. Quando un browser preleva una pagina Web, effettua prima una connessione TCP per ottenere il testo della pagina (in linguaggio HTML), quindi chiude la connessione, ed esamina la pagina. Se ci sono figure o icone, effettua una connessione TCP separata, per prelevarle. Si suggeriscono due progetti alternativi per migliorare le prestazioni.
33. Quando si usa la semantica di sessione, è sempre vero che i cambiamenti ad un file sono immediatamente visibili al processo che li ha effettuati, e non lo sono mai ai processi sulle altre macchine. In ogni caso, è un problema aperto se debbano o meno esser immediatamente visibili agli altri processi sulla stessa macchina. Si discutano entrambe le possibilità.
34. In AFS interi file sono messi nella cache delle macchine client. Si supponga di aver allocato dello spazio disco per i file da mettere nella cache, e che tale spazio sia pieno. Quando viene richiesto un nuovo file, cosa si dovrebbe fare? Si proponga un algoritmo.
35. Quando processi multipli devono accedere ai dati, in che modo l'accesso basato sugli oggetti è preferibile alla memoria condivisa?
36. Quando un'operazione *in* di Linda viene effettuata per localizzare una tupla, ricerare linearmente nell'intero spazio delle tuple è molto inefficiente. Si progetti un modo per organizzare lo spazio delle tuple che acceleri le ricerche in tutte le operazioni *in*.
37. La copia dei buffer richiede tempo. Si scriva un programma in C, per trovare quanto tempo richieda su un sistema cui si abbia accesso. Si usino le funzioni *clock* o *times* per determinare quanto tempo richieda la copia di un grande array. Si faccia il test con diverse dimensioni di array, per separare il tempo di copia da quello dovuto all'overhead.
38. Si scrivano funzioni C che potrebbero servire come stub client-server, per fare una chiamata RPC alla funzione standard *printf*, e un programma principale per testare le funzioni. Il client e il server dovrebbero comunicare per mezzo di una struttura dati che si possa trasmettere in rete. Si possono impostare limiti ragionevoli alla lunghezza della stringa di formato, e al numero, tipo e dimensioni delle variabili accettate dallo stub del client.
39. Si scrivano due programmi per simulare il bilanciamento del carico su un multi-computer. Il primo dovrebbe inizializzare  $m$  processi distribuiti su  $n$  macchine, secondo un file di inizializzazione. Ciascun processo dovrebbe avere un tempo di esecuzione scelto casualmente da una distribuzione gaussiana, la cui media e deviazione standard siano parametri della simulazione. Alla fine di ciascuna esecuzione il processo crea un certo numero di nuovi processi, scelto da una distribuzione di Poisson. Quando un processo finisce, la CPU deve decidere se sbarazzarsi di processi, o provare a trovarne di nuovi. Il primo programma dovrebbe usare l'algoritmo iniziato dal mittente per liberarsi di lavoro, se ha più di  $k$  processi totali sulla sua macchina. Il secondo programma dovrebbe usare l'algoritmo iniziato dal ricevente per prelevare lavoro quando necessario. Si faccia ogni altra ragionevole ipotesi necessaria, ma la si enunci chiaramente.



# SICUREZZA

Molte aziende possiedono informazioni preziose, che devono essere strettamente riservate: queste informazioni possono essere tecniche (ad esempio il progetto di un nuovo chip o di un nuovo software), commerciali (ad esempio studi di competitività o piani di marketing), finanziarie (ad esempio progetti per un'offerta di azioni), legali (ad esempio documenti relativi ad una potenziale fusione od acquisizione) e così via. Spesso queste informazioni sono protette da una guardia in uniforme, posta all'ingresso dell'edificio, che controlla che tutte le persone che accedono a tale struttura siano munite di un cartellino di riconoscimento appropriato. Inoltre, molti uffici possono essere chiusi a chiave, così come alcuni schedari, al fine di garantire che solo le persone autorizzate possano avere accesso a tali informazioni.

Siccome molte di queste informazioni sono immagazzinate in computer, il bisogno di proteggerle è divenuto sempre più importante; pertanto una grande preoccupazione di tutti i sistemi operativi è proteggere tali informazioni da un uso non autorizzato. Sfortunatamente ciò sta diventando sempre più difficile perché diffusamente si accetta come un fenomeno normale ed accettabile che il sistema sia "gonfio". Nelle sezioni che seguono affronteremo diversi problemi relativi a sicurezza e protezione, alcuni dei quali hanno analogie con la protezione di informazioni su carta che avviene nel mondo reale, mentre altri sono caratteristici dei computer. In questo capitolo esamineremo come la sicurezza per i computer venga applicata ai sistemi operativi.

## 9.1 L'ambiente di sicurezza

I termini "sicurezza" e "protezione" sono spesso utilizzati in maniera intercambiabile; nonostante ciò, spesso è utile fare una distinzione: da una parte, ci sono i problemi generali che si incontrano se ci si vuole garantire che i file non siano letti o modificati da persone non autorizzate (il che comprende problemi tecnici, amministrativi, legali e politici), dall'altra parte, ci sono i meccanismi specifici usati dal sistema operativo per offrire la sicurezza. Per evitare confusione si userà il termine **sicurezza** (*security*) quando ci si riferisce al problema generale, e il termine **meccanismi di protezione** quando ci si riferisce a meccanismi specifici usati dal sistema operativo per salvaguardare le informazioni memorizzate nel computer, anche se il confine tra di essi non è ben definito. Dapprima si tratterà la sicurezza, per vedere quale sia la natura del problema; più avanti nel capitolo tratteremo i meccanismi di protezione, ed i modelli utilizzabili per ottenere la sicurezza.

La sicurezza ha molti aspetti, tra cui i tre più importanti sono le minacce, gli intrusi e la perdita accidentale di dati; questi verranno esaminati uno alla volta.

### 9.1.1 Minacce

Dal punto di vista della sicurezza, i sistemi basati su computer hanno tre principali obiettivi, a cui corrispondono differenti minacce, come mostrato in Figura 9.1. Il primo è la **confidenzialità dei dati**, che consiste nel mantenere tali informazioni segrete. Più dettagliatamente, se il proprietario di certe informazioni ha deciso che tali dati siano resi disponibili solo ad alcune persone e non ad altre, il sistema dovrebbe garantire che utenti non autorizzati non possano usufruire di tali informazioni; o quantomeno il proprietario delle informazioni dovrebbe potere specificare chi può vedere cosa, ed il sistema dovrebbe fare rispettare tali restrizioni.

Il secondo obiettivo è l'**integrità dei dati**: gli utenti non autorizzati non dovrebbero essere in grado di modificare nessun dato senza il permesso del proprietario. Da questo punto di vista, inodificare dati comprende non solo cambiare i dati, ma anche rimuovere alcune informazioni, ed aggiungerne altre false. Pertanto se un sistema non può garantire che i dati in esso immagazzinati rimangano immutati sino a decisione contraria del proprietario, tale sistema informativo risulta inutile!

Il terzo obiettivo è la **disponibilità del sistema**: nessuno può disturbare il sistema rendendolo inutilizzabile. Purtroppo gli attacchi tesi al **rifiuto del servizio** sono sempre più frequenti: ad esempio, mandare un numero eccessivo di richieste ad un computer usato come server per Internet, può bloccarlo, visto che quasi tutto il suo tempo di CPU viene utilizzato per esaminare e scartare le richieste che arrivano: se per elaborare una richiesta di lettura di una pagina Web servono ad esempio 100 microsecondi, allora chiunque sia in grado di mandare 10000 richieste al secondo potrà annientare tale server. Attualmente esistono modelli e tecnologie in grado di affrontare attacchi alla confidenzialità ed all'integrità

| Obiettivo                 | Minaccia              |
|---------------------------|-----------------------|
| Confidenzialità dei dati  | Esposizione dei dati  |
| Integrità dei dati        | Manomissione dei dati |
| Disponibilità del sistema | Rifiuto del servizio  |

Figura 9.1 Sicurezza: obiettivi e minacce.

dei dati, al contrario evitare attacchi che mirano al rifiuto del servizio del sistema è molto più difficile.

Un altro aspetto della sicurezza è la **privacy**, ossia proteggere gli utenti dall'uso improprio di informazioni circa la loro persona, un problema che implica anche aspetti legali e morali. Un governo dovrebbe compilare dossier su ognuno al fine di scoprire chi non versi i contributi o chi evada la tasse, a seconda delle idee politiche? La polizia dovrebbe controllare tutto di tutti al solo fine di debellare il crimine organizzato? Quali sono i diritti dei datori di lavoro e delle compagnie di assicurazioni, e cosa accade quando questi diritti vanno in contrasto con i diritti degli individui? Tutti questi aspetti sono molto importanti, ma vanno oltre lo scopo di questo di libro.

## 9.1.2 Intrusi

La maggior parte della gente è civile ed obbedisce alle leggi, perché dunque bisogna preoccuparsi della sicurezza? Perché sfortunatamente esistono persone molto poco civili, il cui scopo è causare problemi (eventualmente per loro profitto personale). Nel gergo della sicurezza, le persone che curiosano dove non dovrebbero, sono dette **intrusi**, o a volte **avversari**. Gli intrusi agiscono principalmente in due modi: gli intrusi passivi vogliono solamente leggere file su cui non hanno diritti di lettura; gli intrusi attivi sono molto più dannosi, infatti vogliono apportare cambiamenti non autorizzati ai dati. Pertanto nel progettare un sistema sicuro contro gli intrusi, è importante tenere a mente il tipo di intruso da cui ci si vuole proteggere. Alcune categorie tipiche sono:

1. Utenti casuali non tecnici. Molte persone hanno PC collegati a sistemi che hanno file condivisi, ed essendo la natura umana quello che è, se non vi sono sbarramenti, qualcuno leggerà la posta elettronica, ed anche i file altrui, se non si frappongono ostacoli. Ad esempio, la maggior parte dei sistemi UNIX da per scontato che tutti i file creati siano pubblici in lettura.
2. Curiosi infiltrati. Studenti, programmatore di sistema, operatori ed altro personale tecnico spesso considerano una sfida personale infrangere la sicurezza del sistema locale: essi sono spesso altamente qualificati e sono disposti a sacrificare una buona quantità del loro tempo in questo sforzo.
3. Tentativi il cui esplicito obiettivo è un vantaggio economico. Alcuni programmatore per applicazioni bancarie hanno tentato di penetrare i sistemi delle banche per cui stavano lavorando: i metodi sono diversi, e vanno dal cambiare il software per troncare piuttosto che arrotondare gli interessi, mantenendo la frazione di centesimo per se stessi, allo svuotare i conti correnti non usati da anni, fino al ricatto ("Pagami o distruggerò tutti gli archivi bancari!").
4. Spionaggio commerciale o militare. Lo spionaggio consiste in un tentativo serio e ben organizzato da parte di un concorrente, o di una nazione straniera, di rubare programmi, documenti segreti, brevetti, tecnologia, progetti di circuiti, piani di mercato e così via. Spesso questo tentativo comporta l'intercettazione di messaggi, o anche l'uso di antenne puntate verso i computer per intercettarne le radiazioni elettromagnetiche.

Dovrebbe essere chiaro che provare ad evitare che un governo straniero rubi segreti militari è un problema molto diverso dall'evitare che studenti inseriscano nel sistema un "messaggio del giorno" divertente. Quindi gli sforzi necessari per sicurezza e protezione dipendono chiaramente da chi si ritiene il nemico.

Un'altra categoria di problemi di sicurezza che si è manifestata negli ultimi anni è il virus e tale argomento verrà lungamente trattato nel seguito: un virus è sostanzialmente un pezzo di codice che è in grado di replicarsi e, di solito, provoca parecchi danni. In un certo qual modo, chi crea il virus è anche un intruso, e spesso di alto profilo tecnico; tuttavia la differenza tra un intruso convenzionale e un virus è che il primo termine si riferisce ad una persona, che sta cercando di penetrare in un sistema per causare danni; mentre il secondo si riferisce ad un programma, scritto da tale persona, e poi immesso nel mondo al fine di causare danni. Gli intrusi cercano di penetrare sistemi specifici (ad esempio quelli appartenenti ad una banca o al Pentagono) per rubare o distruggere particolari informazioni, al contrario il virus di solito causa danni non specifici. Da un certo punto di vista, l'intruso è come qualcuno munito di pistola che cerchi di uccidere una persona in particolare, mentre il creatore del virus assomiglia ad un terrorista che piazza una bomba per uccidere molte persone, senza alcuna preferenza per qualcuna in particolare.

### 9.1.3 Perdita accidentale di dati

Oltre alle minacce dovute a intrusi, si possono perdere informazioni preziose anche per caso. Alcune delle più comuni cause di perdite di dati accidentali sono:

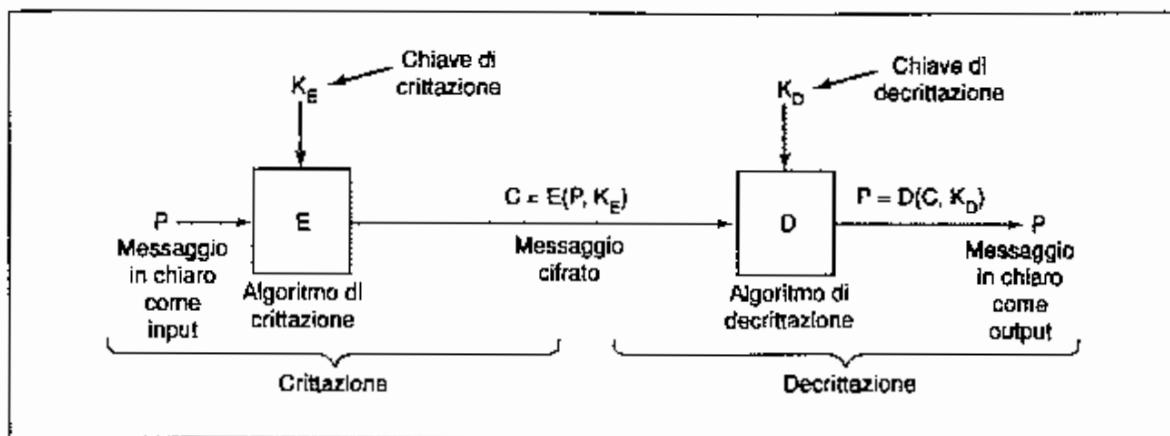
1. Cause di forza maggiore: incendi, alluvioni, terremoti, guerre, rivolte o topi che rovistano nastri o floppy disk.
2. Errori hardware o software: malfunzionamenti della CPU, nastri o dischi illeggibili, errori di telecomunicazione, errori nei programmi.
3. Errori umani: input scorretti, errato montaggio di dischi o nastri, esecuzione del programma sbagliato, smarrimento di dischi o nastri, o altri sbagli.

Alla maggior parte di queste cause si può ovviare con una adeguata politica di backup, preferibilmente tenendo tali dati lontano dagli originali! Infine se proteggere i dati da perdite accidentali può sembrare un compito stupido, rispetto al proteggersi dall'intrusione, all'atto pratico si riscontra che, di solito, i maggiori danni sono dovuti proprio a tali cause accidentali.

## 9.2 Principi di crittografia

Anche se una seria disamina sulla crittografia va oltre lo scopo di questo libro, una minima conoscenza su tale argomento può risultare utile per comprendere parti di questo capitolo e di alcuni successivi. Molti ottimi libri incentrati sulla sicurezza dei computer analizzano a fondo questo argomento; il lettore più interessato può fare riferimento ad essi (ad esempio Kaufman et al., 1995, e Pfleeger, 1997). Nel seguito introduciamo brevemente i principi della crittografia per chi fosse digiuno di tale argomento.

L'obiettivo della crittografia è di prendere un messaggio o un file, chiamato **messaggio in chiaro** (plaintext), e crittarlo in un **messaggio cifrato** (ciphertext), in modo che solo le persone autorizzate sappiano come decrittarlo e ottenere nuovamente il messaggio originale; e per tutti gli altri il messaggio cifrato rimanga una sequenza incomprensibile di bit. Può sembrare strano a chi non è addentro a questo argomento, ma gli algoritmi (funzioni) di crittazione e decrittazione dovrebbero essere sempre pubblici: tentare di tenerli segreti non funziona mai, e dà alle persone che stanno cercando di mantenere la segretezza un fal-



**Figura 9.2** Relazione tra messaggio in chiaro e messaggio cifrato.

so senso di sicurezza. In campo commerciale questo modo di agire è chiamato **sicurezza per misconoscenza** ed è utilizzato solo da chi è neofita nel campo della sicurezza! Tuttavia, in maniera abbastanza curiosa, questa categoria comprende anche molte grandi multinazionali che dovrebbero conoscere meglio tale problema.

Al contrario, la segretezza dipende dai parametri dell'algoritmo chiamati **chiavi**: se  $P$  è il messaggio in chiaro e  $K_E$  la chiave di crittazione,  $C$  il messaggio cifrato,  $E$  l'algoritmo di crittazione, allora  $C = E(P, K_E)$ . Quella appena data è la definizione di crittazione e significa che il messaggio cifrato è ottenuto usando l'algoritmo di crittazione (noto),  $E$ , utilizzando un messaggio in chiaro,  $P$ , ed una chiave di crittazione (non nota),  $K_E$ , come suoi parametri.

In maniera del tutto analoga vale  $P = D(C, K_D)$ , dove  $D$  è l'algoritmo di decrittazione e  $K_D$  è la chiave di decrittazione. Come prima, questo significa che per riottenere il messaggio in chiaro,  $P$ , a partire dal messaggio cifrato,  $C$ , e dalla chiave di decrittazione,  $K_D$ , bisogna utilizzare l'algoritmo  $D$ , con  $C$  e  $K_D$  come suoi parametri. Le relazioni tra i vari componenti vengono illustrate nella Figura 9.2.

## 9.2.1 Crittografia a chiave privata

Per chiarire quanto detto, consideriamo un algoritmo di crittazione in cui ogni lettera è sostituita da un'altra, ad esempio, la *A* diventa *Q*, la *B* diventa *W*, la *C* diventa *E* e così via per tutte le altre lettere come segue:

messaggio in chiaro:  
messaggio cifrato:

ABCDEFGHIJKLMNOPQRSTUVWXYZ  
QWERTYUIOPASDFGHJKLZXCVBNM

Questo generico sistema di crittazione è chiamato **sostituzione monoalfabetica**, e la chiave è rappresentata dalla stringa di 26 lettere corrispondente all'alfabeto inglese. Nell'esempio visto, la chiave è *QWERTYUIOPASDFGHJKLZXCVBNM* e, data tale chiave, la parola *ATTACCO* verrebbe crittata in *QZZQEEG*. La chiave di decrittazione, che permette di ottenere nuovamente il messaggio in chiaro, partendo da quello cifrato, è nel nostro esempio *KXVMCNOPHQRSZYIJADLEGWBUFT*, in quanto alla *A* nel messaggio cifrato corrisponde la *K* nel messaggio in chiaro, alla *B* corrisponde la *X*, e così via.

A prima vista, questo potrebbe apparire un sistema sicuro perché, sebbene il critto-analista conosca il sistema in generale (sostituzione lettera per lettera), non conosce quale delle  $26! \sim 4 \times 10^{26}$  possibili chiavi sia stata utilizzata. Ciononostante, dato un numero di

messaggi cifrati sorprendentemente basso, il sistema può essere facilmente decifrato. L'attacco più semplice si avvantaggia delle proprietà statistiche dei linguaggi naturali: in inglese, ad esempio, la lettera più comune è la *e*, seguita da *t*, *o*, *a*, *n*, e così via, mentre i gruppi di due lettere più comuni (chiamati **digrammi**) sono *th*, *in*, *er*, *re*. Utilizzando perciò tale conoscenza, il sistema può essere facilmente decifrato.

Molti sistemi crittografici come quello appena descritto hanno la proprietà che data una chiave di crittazione è facile trovare la chiave di decrittazione e viceversa; tali sistemi vengono detti **crittografia a chiave privata o a chiave simmetrica**. Pertanto, sebbene la sostituzione monoalfabetica sia priva di valore, esistono altri algoritmi a chiave simmetrica che si sono dimostrati relativamente sicuri se le chiavi utilizzate sono abbastanza lunghe. Per essere davvero sicuri si dovrebbero utilizzare probabilmente chiavi a 1024 bit, ottenendo quindi  $2^{1024} \sim 2 \times (10^{308})$  chiavi possibili; naturalmente chiavi più brevi possono contrastare i dilettanti, ma non i governi stranieri.

## 9.2.2 Crittografia a chiave pubblica

I sistemi a chiave privata sono efficienti perché il numero di calcoli richiesti per crittare o decrittare un messaggio è gestibile, ma hanno anche un grande inconveniente: il mittente e il ricevente devono essere entrambi a conoscenza della chiave privata che condizionano, e probabilmente si devono incontrare fisicamente per scambiarsi tale chiave. Per aggirare tale ostacolo, viene utilizzata la **crittografia a chiave pubblica** (Diffie e Hellman, 1976). Questo sistema è caratterizzato dall'uso di chiavi distinte per crittazione e decrittazione, e data una chiave pubblica di crittazione ben scelta, è praticamente impossibile risalire alla corrispondente chiave di decrittazione. A questo punto la chiave di crittazione può essere resa pubblica mentre la chiave privata di decrittazione rimane segreta.

Per dare un'idea di cosa si intenda per crittografia a chiave pubblica, consideriamo le seguenti domande:

- Domanda 1: Quanto fa  $314159265358979 \times 314159265358979$ ?  
 Domanda 2: Qual è la radice quadrata di  
 $3912571506419387090594828508241$ ?

La maggior parte degli studenti delle medie, data loro una matita, un pezzo di carta su cui scrivere, e la promessa di essere ricompensati con un gelato davvero enorme per una risposta esatta, potranno risolvere il primo quesito in un'ora o due; in maniera analoga, la maggior parte degli adulti, data loro una matita, un pezzo di carta su cui scrivere, e la promessa di una riduzione del 50% delle tasse per il resto della loro vita, non potranno davvero risolvere il secondo problema senza utilizzare una calcolatrice, un computer o un qualsiasi altro aiuto esterno. Sebbene l'elevazione al quadrato e l'estrazione della radice quadrata siano operazioni inverse, differiscono enormemente nella loro complessità di calcolo: questa asimmetria forma la base della crittografia a base pubblica. La crittazione utilizza operazioni semplici, ma la decrittazione senza chiave richiede numerose e complicate operazioni.

Ad esempio, un sistema a chiave pubblica chiamato **RSA** sfrutta il fatto che moltiplicare grandi numeri sia molto più facile che fattorizzarli, specialmente quando tutta l'aritmetica è fatta utilizzando l'aritmetica modulo, e tutti i numeri coinvolti sono composti da centinaia di cifre (Rivest et al., 1978). Questo sistema viene utilizzato diffusamente nel mondo della crittografia, ed analogamente si utilizzano sistemi basati su logaritmi discreti (El Gamal, 1985); tuttavia il principale problema della crittografia a chiave pubblica è che, rispetto alla crittografia a chiave simmetrica, risulta migliaia di volte più lento!

La crittografia a chiave pubblica è strutturata nel modo seguente: si prende una copia di chiavi (chiave pubblica, chiave privata) e si rende nota quella pubblica; la chiave pubblica rappresenta la chiave di crittazione, mentre la chiave privata rappresenta la chiave di decrittazione. Inoltre di solito la generazione delle chiavi è automatica, eventualmente ottenuta da una password scelta dall'utente ed inserita come "seme" dell'algoritmo. Pertanto per spedire un messaggio segreto ad un utente, il mittente critta il messaggio con la chiave pubblica del ricevente, e poiché solo tale persona possiede la chiave privata, sarà il solo a poter decrittare il messaggio.

### 9.2.3 Funzioni unidirezionali

Esistono diverse situazioni, che esamineremo nel seguito, nelle quali è auspicabile avere alcune funzioni,  $f$ , per le quali, date  $f$  ed il suo parametro  $x$ , sia facile calcolare  $y = f(x)$ , mentre, data  $f(x)$ , trovare  $x$  sia praticamente impossibile. Tali funzioni tipicamente manipolano i bit in un modo complicato: si può cominciare dall'inizializzare  $y$  ad  $x$ , poi si può iterare un ciclo tante volte quanti sono i bit a 1 presenti in  $x$ , permutando ad ogni iterazione i bit di  $y$  in maniera dipendente dall'iterazione stessa, sommandovi una costante diversa ad ogni iterazione, e in generale mischiando a fondo i bit.

### 9.2.4 Firme digitali

Spesso si ha la necessità di una firma digitale su un documento; ad esempio, supponiamo che il cliente di una banca incarichi questa di comprare alcune azioni, tramite una e-mail, e che un'ora dopo che il messaggio è stato spedito e l'ordine eseguito, il titolo crolli. A questo punto il cliente può negare di avere spedito tale e-mail, ed anche se la banca può mostrarla, il cliente può sostenere che la banca l'abbia contrattata per ottenere la commissione. Come può un giudice capire chi stia dicendo la verità?

La firma digitale rende possibile firmare e-mail ed altri documenti in formato digitale, in modo che questi non possano venire disconosciuti da chi li ha precedentemente spediti. Una maniera molto comune per realizzare ciò è far passare il documento attraverso un algoritmo di hashing unidirezionale che è molto difficile da invertire. La funzione di hashing tipicamente produce un risultato di lunghezza fissa indipendente dalla dimensione del documento originale. Le più famose funzioni di hashing sono MD5 (Message Digest) che produce un risultato a 16 byte (Rivest, 1992), e SHA (Secure Hash Algorithm) che produce un risultato a 20 byte (NIST, 1995).

Il passo successivo consiste nell'utilizzare la crittografia a chiave pubblica come

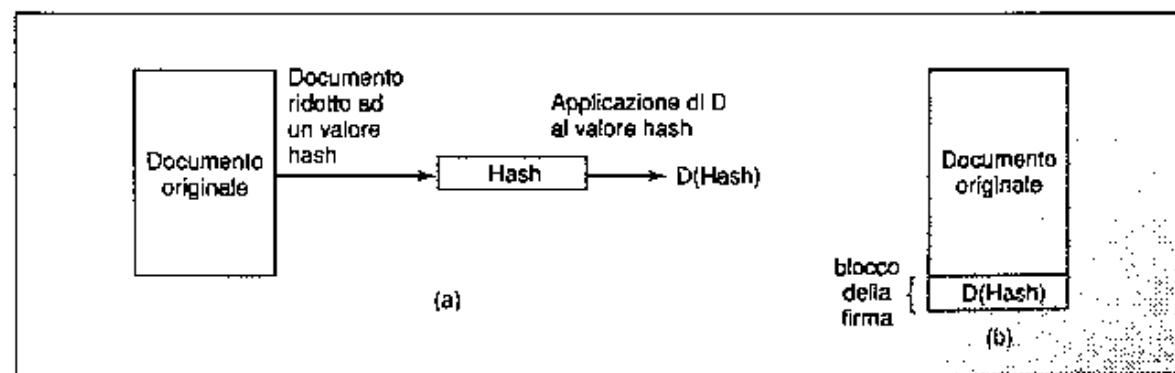


Figura 9.3 (a) Calcolo del blocco della firma. (b) Ciò che appare al ricevente

descritto in precedenza. Il proprietario del documento applica quindi la sua chiave privata all'hash per ottenere  $D(\text{hash})$ : questo valore, chiamato **blocco della firma**, viene allegato al documento e spedito al ricevente, come mostrato in Figura 9.3. Qualche volta ci si riferisce all'applicazione di  $D$  all'hash come fosse una decrittazione dell'hash stesso, ma in realtà non vi è alcuna decrittazione in quanto l'hash non è stato crittato, ma è stata solamente operata una trasformazione matematica su questo.

Quando il documento e l'hash giungono al destinatario, questi dapprima calcola l'hash del documento utilizzando MD5 o SHA, come era stato stabilito in precedenza, poi applica la chiave pubblica del mittente al blocco della firma per ottenere  $E(D(\text{hash}))$ . In effetti, questa operazione critta l'hash decrittato, annullandone reciprocamente l'effetto e riottenendo l'hash. Se l'hash calcolato non corrisponde a quello del blocco della firma, significa che il documento, il blocco della firma o entrambi sono stati manomessi (o accidentalmente modificati). Il valore di questo schema è che esso applica la (lenta) crittografia a chiave pubblica solo ad una parte relativamente piccola dei dati, l'hash. Sottolineiamo che questo metodo funziona se per tutte le  $x$  vale

$$E(D(x)) = x$$

Inoltre non è garantito a priori che tutte le funzioni godranno di questa proprietà, perché quello che era stato richiesto all'inizio era che valesse

$$D(E(x)) = x$$

cioè, che  $E$  fosse la funzione di crittazione e che  $D$  fosse quella di decrittazione. Per ottenere oltre a questo, anche la proprietà della firma, l'ordine con cui si eseguono le applicazioni non deve essere importante, ossia le funzioni  $E$  e  $D$  devono essere commutative. Fortunatamente l'algoritmo RSA gode di questa proprietà.

Per utilizzare questo schema per la firma digitale, il ricevente deve conoscere la chiave pubblica del mittente: alcuni utenti inseriscono la loro chiave pubblica nelle loro home-page; altri invece non agiscono in tale maniera, per il timore che un intruso penetri illegalmente in esse e modifichi le loro chiavi. Un metodo diffuso tra chi spedisce messaggi è quello di allegare al messaggio un certificato che contenga il nome dell'utente, la chiave pubblica e sia firmato digitalmente da una terza persona di fiducia. Una volta che l'utente abbia acquisito la chiave pubblica di tale persona fidata, può accettare i certificati di tutti quei mittenti che utilizzino questa persona fidata per ottenere i loro certificati.

Abbiamo quindi descritto come la crittografia a chiave pubblica possa essere utilizzata per le firme digitali; occorre però ricordare che esistono anche degli schemi che non coinvolgono la crittografia a chiave pubblica.

### 9.3 Autenticazione dell'utente

Ora che abbiamo qualche conoscenza di crittografia, cominciamo a dare un'occhiata al problema della sicurezza nei sistemi operativi. Quando un utente si collega ad un computer, il sistema operativo normalmente desidera sapere chi sia l'utente: questo processo è chiamato **autenticazione dell'utente**.

L'autenticazione dell'utente è uno di quegli aspetti che intendevamo con "l'ontogenesi riassume la filogenesi" nella Sezione 1.2.5. I primi mainframe, come ENIAC ad esem-

pio, non avevano un sistema operativo, tanto meno una procedura di login; i mainframe successivi ed i sistemi time-sharing avevano una procedura di login per l'autenticazione di processi ed utenti.

I primi minicomputer (ad esempio PDP-1 e PDP-8) non avevano una procedura di login, ma con la diffusione di UNIX sui minicomputer PDP-11, il problema dell'effettuare il login si riproponeva. I primi personal computer (ad esempio Apple II e il PC IBM delle origini) non avevano una procedura di login, ma i sistemi operativi di personal computer più sofisticati, quali Windows 2000, richiedono nuovamente una procedura di login sicura; inoltre l'utilizzo di un personal computer per accedere ad un server di una rete locale (LAN) o ad un qualche account di un sito Web di commercio elettronico richiede sempre la procedura di login. Pertanto il tema del login sicuro ha attraversato diverse fasi, ed è ancora una volta un tema importante.

Una volta che è stato determinato che l'autenticazione è un problema rilevante, il passo successivo è trovare un buon modo per realizzarla. La maggior parte dei metodi per identificare l'utente quando questi sta tentando di eseguire il login, sono basati su tre principi generali, e cioè sull'identificare:

1. Qualcosa che l'utente conosce.
2. Qualcosa che l'utente ha.
3. Qualcosa che l'utente è.

Questi principi portano a schemi di autenticazione diversi, con complessità diverse e proprietà di sicurezza differenti, che esamineremo uno alla volta nelle sezioni successive.

Le persone intenzionate a causare problemi ad un particolare sistema devono prima effettuare il login in quel sistema, il che significa andare oltre qualsiasi procedura d'autenticazione utilizzata. Nella stampa popolare queste persone vengono chiamate **hacker**, anche se nel mondo dell'informatica, "hacker" è un termine onorifico, riservato ai più abili programmati: solo alcuni di questi sono dei mascalzoni, mentre la maggior parte non lo è, ma purtroppo la stampa ha tratto l'accezione negativa del termine. Al contrario, per rispetto ai veri hacker, noi utilizzeremo l'accezione originale di tale termine e chiameremo **cracker** quelle persone che cercano di penetrare i sistemi in maniera illegale.

### 9.3.1 Autenticazione tramite password

La forma di autenticazione più diffusa è quella di richiedere all'utente di digitare la sua login e la sua password. La protezione tramite password è facile sia da capire che da implementare; l'implementazione più semplice consiste nel mantenere in memoria una lista di coppie (login, password): la login digitata viene ricercata nella lista e la password confrontata con quella ivi presente; se queste ultime corrispondono, l'accesso è consentito; se non corrispondono, l'accesso viene rifiutato.

Non c'è bisogno di precisare che, mentre la password viene digitata, il computer non dovrebbe visualizzarla sullo schermo, per salvaguardarla da occhi indiscreti. In Windows 2000 ad ogni carattere digitato, appare un asterisco; in UNIX invece non compare nulla sullo schermo quando la password viene digitata. Lo schema di Windows 2000 può aiutare gli utenti distratti a sapere quanti caratteri abbiano digitato fino a quel momento, ma allo stesso tempo rivela la lunghezza della password agli "origliatori" (il termine etimologicamente più corretto, "guardone", non ci sembra utilizzabile in questo contesto). Insomma, dal punto di vista della sicurezza, il silenzio è d'oro!

Un altro caso in cui non affrontare correttamente tale problema ha implicazioni sulla

|                                                    |                                              |                                                              |
|----------------------------------------------------|----------------------------------------------|--------------------------------------------------------------|
| LOGIN: ken<br>PASSWORD: FooBar<br>SUCCESSFUL LOGIN | LOGIN: carol<br>INVALID LOGIN NAME<br>LOGIN: | LOGIN: carol<br>PASSWORD: Indunno<br>INVALID LOGIN<br>LOGIN: |
| (a)                                                | (b)                                          | (c)                                                          |

**Figura 9.4** [a] Login effettuata con successo. [b] Accesso rifiutato dopo l'inserimento del nome. [c] Accesso rifiutato dopo l'inserimento di nome e password.

sicurezza è mostrato in Figura 9.4. Nella Figura 9.4(a) viene mostrato un login effettuato con successo, ove l'output del sistema è visualizzato a lettere maiuscole, mentre l'input dell'utente a lettere minuscole; nella Figura 9.4(b) viene invece mostrato il tentativo fallito di un cracker di penetrare il sistema A; infine nella Figura 9.4(c) viene illustrato il tentativo fallito di un cracker di penetrare il sistema B.

Nella Figura 9.4 (b) il sistema reagisce appena rileva un nome non valido per il login: questo è un errore, in quanto permette al cracker di continuare a tentare con più nomi fino a trovarne uno valido. Nella Figura 9.4 (c) invece, viene chiesta sempre la password al cracker, così egli non può sapere se il nome inserito come login sia valido o meno: tutto ciò che il cracker scopre è che la combinazione nome-password che ha provato è scorretta.

## Come penetrano i cracker

La maggior parte dei cracker penetrano i sistemi collegandosi al computer scelto come bersaglio e tentando molte combinazioni (login, password) fino a trovarne una che funziona. Molte persone utilizzano il loro nome in una qualche forma come loro login; ad esempio per Ellen Ann Smith, ellen, smith, ellen\_smith, ellen-smith, ellen.smith, esmith, easmith, eas, eccetera, sono tutti candidati ragionevoli. Pertanto, munito di uno di quei libri intitolati "4096 nomi per il vostro bambino", e di un elenco telefonico pieno di cognomi, un cracker può facilmente redigere una lista computerizzata delle potenziali login a seconda del paese attaccato (ellen\_smith potrebbe andare bene negli USA o in Gran Bretagna, ma probabilmente non in Giappone!).

Naturalmente azzeccare la login non è sufficiente, bisogna indovinare anche la password. Ma quanto è difficile trovare la password? Più facile di quanto si possa pensare: Morris e Thompson (1979) fecero uno studio sulla sicurezza delle password dei sistemi UNIX, compilando una lista di password probabili: nomi, cognomi, indirizzi, città, parole prese da un piccolo dizionario (anche scritte a rovescio), numeri di targa ancora validi e brevi stringhe di caratteri casuali. Controllarono quindi se qualcuna delle password degli utenti corrispondesse a qualcuna della loro lista: un buon 86% delle password risultò comparire nella loro lista! Un risultato simile fu poi ottenuto anche da Klein (1990).

Si può pensare che utenti più capaci sceglieranno password più complesse, ma in realtà non lo faranno. Uno studio del 1997 effettuato sulle password utilizzate nel distretto finanziario di Londra ha rivelato che l'82% di queste potrebbe essere facilmente individuato: le password comunemente utilizzate erano termini sessuali, parolacce, nomi di persone (spesso quello di un familiare o un campione dello sport), destinazioni di vacanze ed oggetti che di solito si possono trovare in ufficio (Kabay, 1997). In questo modo un cracker può compilare una lista di potenziali login ed una lista di potenziali password senza molto affanno.

Ma è davvero così importante che le password si possano individuare? La risposta è

sì! Nel 1998 il *San Jose Mercury News* riferì che un tizio residente a Berkeley, Peter Shipley, aveva adibito parecchi computer dunque a war dialers, ossia macchine che componevano tutti i 10000 numeri di telefono appartenenti ad un centralino (ad esempio (415) 770xxxx), di solito in ordine casuale, per impedire alle compagnie telefoniche di rilevare l'origine di un tale sovraccarico. Dopo 2,6 milioni di telefonate, costui aveva localizzato 20000 computer nella Bay Area, di cui 200 non avevano alcun sistema di sicurezza; egli calcolò inoltre che un cracker determinato sarebbe potuto penetrare in circa il 75% degli altri (Denning, 1999).

La combinazione di un war dialer e della facilità nell'individuare password può essere letale: un cracker australiano ha scritto un programma che componeva sistematicamente tutti i numeri di un centralino telefonico, quindi cercava di penetrare utilizzando password casuali ed infine gli notificava i successi riscontrati. Tra i molti sistemi in cui riuscì a penetrare vi fu il computer della Citibank in Arabia Saudita; ciò gli permise di ottenere numeri di carte di credito, limiti di credito (in un caso 5 milioni di dollari!) e registrazioni di transazioni (tra cui almeno una visita ad un bordello). Anche un suo collega, cracker anch'esso, penetrò nella banca e si impossessò di 4000 numeri di carte di credito (Denning, 1999). Se fosse stato fatto un cattivo uso di tali informazioni, la banca avrebbe negato senza dubbio alcuno e con vigore che vi potesse essere stato un errore, sostenendo anzi che tale informazione doveva essere stata rivelata dal cliente.

Un'alternativa all'utilizzo di un war dialer è attaccare i computer in Internet. Ogni computer collegato ad Internet è identificato da un indirizzo IP a 32 bit; di solito questi indirizzi sono scritti in notazione decimale con punti, ad es. w.x.y.z, dove ognuna delle quattro componenti dell'indirizzo IP è un intero tra 0 e 255 (in base dieci). Un cracker può sapere facilmente se esiste un qualche computer che ha questo indirizzo IP, che è attivo e funziona, digitando solamente

`ping w.x.y.z`

Se il computer è vivo (attivo), risponde ed il programma ping renderà noto quanti millisecondi è durata la trasmissione del segnale tra andata e ritorno (anche se alcuni siti ora stanno disabilitando il ping per prevenire questo tipo di attacco). È facile scrivere un programma che effettui sistematicamente il ping di un gran numero di indirizzi IP, in maniera analoga a come agiscono i war dialer. Se viene scovato un computer attivo all'indirizzo w.x.y.z, il cracker può cercare di penetrarlo col comando

`telnet w.x.y.z`

Se il tentativo di connessione è accettato (il che può anche non accadere, perché non tutti gli amministratori di sistemi accettano login casuali in Internet), il cracker può iniziare i suoi tentativi con login e password delle sue liste. All'inizio saranno solo tentativi ed errori; tuttavia, il cracker può essere prima o poi in grado di penetrare una volta o l'altra nel sistema, e di carpire il file delle password (che nei sistemi UNIX è situato nella directory `/etc/passwd` e spesso è leggibile per tutti gli utenti). Quindi inizierà a immagazzinare informazioni statistiche sulla frequenza di utilizzo delle login per ottimizzare ricerche future.

Molti demoni di telnet interrompono la connessione TCP che sta alla base, dopo un certo numero di tentativi falliti di fare login, per rallentare il lavoro dei cracker. A questo i cracker reagiscono avviando molti processi in parallelo, che lavorano su differenti macchine bersaglio simultaneamente: il loro obiettivo è fare tanti tentativi al secondo quanto ne permetta la banda in uscita. Dal loro punto di vista infatti doverli ripartire su più macchine attaccate simultaneamente non è un vero svantaggio.

Invece di fare il ping su dei computer in ordine di indirizzo IP, il cracker può avere come bersaglio una specifica azienda, un'università, o altre organizzazioni, ad esempio l'Università di Foobar, identificata da *foobar.edu*. Per scoprire quale sia il loro indirizzo IP, quello che devono fare è digitare

```
dnsquery foobar.edu
```

ed otterranno una lista di alcuni dei loro indirizzi IP (in maniera alternativa si possono utilizzare i programmi *nslookup* o *dig*). Poiché molte organizzazioni hanno 65536 indirizzi IP consecutivi (una unità di allocazione molto comune nel passato), una volta che siano noti i primi 2 byte dei loro indirizzi IP (forniti da *dnsquery*), è semplice fare il ping a tutti e 65536 per vedere quali rispondano e quali accettino connessioni via telnet. Da questo punto in avanti, si ritorna a tentare combinazioni di login e password, come abbiamo già discusso.

Non c'è bisogno di dirlo, ma tutto il processo di ricerca del nome del dominio, rilevazione dei primi 2 byte dei suoi indirizzi IP, del fare ping verso questi per controllare quali siano attivi e quali accettino una connessione via telnet, e poi il provare le coppie (login, password) statisticamente probabili, è un processo che si presta facilmente all'automaticazione. Ci vorranno davvero moltissimi tentativi per penetrare, ma se c'è una cosa in cui i computer sono bravi è proprio la ripetizione della stessa sequenza di comandi più e più volte fino ad ottenere quello che si voleva! Un cracker con una connessione via cavo ad alta velocità o DSL può programmare il processo di intrusione in modo che lavori tutto il giorno, e controllare occasionalmente quello che è accaduto.

Chiaramente un attacco via telnet è migliore di un attacco con war dialer, perché è molto più veloce (non si perde tempo nel comporre il numero) ed è molto più economico (non vi sono tariffe telefoniche extraurbane), ma funziona solo con macchine che siano connesse ad Internet e che accettino connessioni via telnet. Nonostante ciò molte aziende (e quasi tutte le università) accettano connessioni via telnet, in modo che gli impiegati in viaggio d'affari o in una differente succursale dell'ufficio (o studenti a casa) possano eseguire un login in remoto.

Non solo le password degli utenti sono spesso poco sicure, a volte anche le password di root lo sono; in particolare alcuni tipi di installazione non insistono affinché venga cambiata la password per difetto con cui il sistema è stato realizzato. Cliff Stoll, un astronomo di Berkeley, aveva osservato delle irregolarità nel suo sistema, pertanto escogitò

```
LBL> telnet elksi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT ABL
```

**Figura 9.5** Come un cracker è riuscito a penetrare un computer del Dipartimento dell'Energia degli Stati Uniti presso il LBL.

una trappola per il cracker che avesse cercato di violare il sistema (Stoll, 1989). Egli osservò la sessione (mostrata in Figura 9.5) digitata da un cracker che era già penetrato in un'altra macchina del Lawrence Berkeley Laboratory (LBL) e stava cercando di violarne un'altra. L'account uucp (UNIX to UNIX Copy Program) viene utilizzata per il traffico di rete tra macchine ed ha modo superuser, in questo modo il cracker si trovava in una macchina del Dipartimento dell'Energia degli Stati Uniti come superuser! Per fortuna il LBL non progetta armi nucleari, come il suo laboratorio gemello a Livermore: si spera che la loro sicurezza sia migliore, ma vi sono poche ragioni per crederlo visto che in un altro laboratorio dedito alla progettazione di armi nucleari, Los Alamos, è stato smarrito un hard disk pieno di informazioni confidenziali, nel corso del 2000.

Una volta che un cracker ha violato un sistema e diventa superuser, può installare un **packet sniffer**, ossia un software che esamina i pacchetti in ingresso o in uscita dalla rete, alla ricerca di particolari stringhe. Fra le stringhe particolarmente interessanti da cercare ci sono quelle delle persone che lavorano sulle macchine compromesse, e che effettuano login in remoto su altre macchine, specialmente se come superuser. Queste informazioni possono essere salvate in un file, che il cracker potrà recuperare con comodo in seguito. In questo modo, un cracker che penetri in una macchina dotata di scarsa sicurezza spesso può utilizzare questo fatto per violare macchine con sicurezza maggiore.

Un numero sempre più grande di violazioni sono realizzate da utenti tecnicamente ingenui che eseguono script recuperati in Internet: questi script o utilizzano gli attacchi brutali precedentemente descritti, oppure cercano di sfruttare i bachi di programmi specifici. I veri hacker si riferiscono ad essi come **script kiddies**.

Di solito lo script kiddie non ha particolari obiettivi o informazioni da rubare, ma cerca macchine in cui sia facile penetrare. Alcuni di questi script scelgono anche di attaccare una rete a caso, utilizzando un numero di rete casuale (nella parte superiore dell'indirizzo IP); quindi esplorano tutte le macchine di tale rete per vedere quali rispondono. Una volta che sia stato acquisito un database di indirizzi IP validi, tutte le macchine vengono attaccate in sequenza. Una conseguenza di questo modus operandi è che può accadere che una nuova macchina di una installazione militare sicura possa essere attaccata poche ore dopo essere stata collegata ad Internet, anche se nessuno a parte l'amministratore sa ancora della sua esistenza.

## Sicurezza delle password in UNIX

Alcuni (vecchi) sistemi operativi memorizzano il file delle password su disco in una forma non criptata, ma protetta dai soliti meccanismi di protezione del sistema. Avere tutte le password in un file su disco in forma non criptata è come andare a cercarsi dei problemi perché troppo spesso molte persone hanno accesso ad esso: tra essi possono esservi amministratori di sistema, operatori di macchina, personale addetto alla manutenzione, programmati, dirigenti e probabilmente anche alcune segretarie.

Una soluzione migliore funziona come segue: il programma che gestisce l'accesso chiede all'utente di digitare login e password; la password viene immediatamente "criptata", utilizzandola come una chiave per criptare un blocco fisso di dati. In effetti viene chiamata in gioco una funzione unidirezionale il cui input è la password ed il cui output è una funzione della password. Questo processo non è una vera crittazione, ma risulta più facile considerarla una crittazione. Il programma che gestisce l'accesso legge quindi il file delle password, che è una serie di linee ASCII, una per ogni utente, fino a trovare la linea contenente il nome dell'utente: se la password (criptata) contenuta in questa linea corrisponde alla password criptata appena calcolata, l'accesso è consentito, altrimenti viene rifiutato. Il vantaggio di questo schema è che nessuno, nemmeno il superuser, può recuperare le password

degli utenti, perché non sono memorizzate da nessuna parte in forma non criptata all'interno del sistema.

Tuttavia anche questo schema può essere attaccato nella maniera seguente: un cracker dapprima crea un dizionario di password possibili come hanno fatto Morris e Thompson; quindi le critta con comodo utilizzando l'algoritmo noto. Quanto tempo porti via questo processo non è rilevante, in quanto viene effettuato prima della violazione vera e propria. A questo punto, munito di una lista copie (password, password crittate), il cracker attacca: legge il file delle password (pubblicamente accessibile) e scopre tutte le password crittate che vengono poi confrontate con quelle della sua lista; perciò ad ogni successo, il nome e la password non crittata sono diventati noti. Un semplice script di shell può automatizzare questo processo, in modo che possa essere portato a termine in una frazione di secondo: un'esecuzione tipica di tale script produrrà dozzine di password.

Riconoscendo la possibilità di tale attacco, Morris e Thompson descrissero una tecnica che rendeva l'attacco quasi inefficace. La loro idea consiste nell'associare un numero casuale di  $n$  bit, chiamato salt (sale), ad ogni password; inoltre il numero casuale cambia ogni volta che la password viene cambiata. Tale numero viene memorizzato nel file delle password in forma non crittata, di modo che tutti lo possano leggere. Quindi, invece di memorizzare la password crittata nel file delle password, la password ed il numero casuale vengono prima concatenati e poi crittati assieme. Il risultato di tale crittazione viene memorizzato nel file delle password, come mostrato in Figura 9.6, per un file delle password comprendente cinque utenti, Bobbie, Tony, Laura, Mark e Deborah. Ad ogni utente corrisponde una linea del file, con tre campi separate da virgole: nome, salt e password + salt crittati. La notazione  $e(Dog4238)$  rappresenta il risultato della concatenazione della password di Bobbie, Dog, col salt casualmente assegnato, 4238, e poi elaborato dalla funzione di crittazione,  $e$ . Ed è proprio il risultato di tale crittazione che viene memorizzato come terzo campo per la linea corrispondente a Bobbie.

Consideriamo ora le implicazioni per un cracker che voglia costruirsi una lista di password possibili, criptarle e poi salvare il risultato in file ordinato,  $f$ , in modo che ogni password crittata possa essere facilmente ritrovata. Se un intruso sospetta che Dog possa essere una password, non è più sufficiente criptare Dog e mettere il risultato in  $f$ ; si devono criptare  $2^n$  stringhe, quali Dog0000, Dog0001, Dog0002, e così via ed inserire il risultato in  $f$ . Questa tecnica accresce la dimensione di  $f$  di  $2^n$ ; UNIX utilizza questo metodo con  $n = 12$ .

Per una sicurezza maggiore, alcune versioni moderne di UNIX rendono il file delle password illeggibile, ma forniscono un programma per cercare dei campi a richiesta, aggiungendo abbastanza ritardo per rallentare enormemente ogni intruso.

La combinazione del salt applicato al file delle password e del renderlo illeggibile se non in maniera indiretta (e lenta) in genere può resistere agli attacchi rivolti ad esso.

|                                  |
|----------------------------------|
| Bobb-e. 4238. e Dog4238          |
| Tony. 2918. e 6%%TaeFF2918       |
| Laura. 6902. e Shakespeare6902   |
| Mark. 1694. e XaBfcBwc1694       |
| Deborah. 1092. e LordByron. 1092 |

Figura 9.6 L'uso di salt per sconfiggere la preelaborazione di password crittate.

## Migliorare la sicurezza delle password

Sebbene l'uso del salt per il file delle password offra una protezione da intrusi che cercino di precalcolare una lista di password crittografate, ottenendo così più password in una volta, non protegge molto un utente *David*, la cui password sia *David*. Ma un cracker può anche cercare di indovinare le password una alla volta e dunque educare gli utenti alla necessità di password più sicure è basilare, ma poche installazioni lo fanno. Un passo in avanti verso l'educazione degli utenti è farli aiutare dal computer: alcuni computer hanno un programma che genera parole casuali senza senso, ma facili da pronunciare, come *fotally*, *garbungy*, *bipitty*, che si possono utilizzare come password (preferibilmente con qualche carattere maiuscolo e speciale inserito nella parola). Inoltre il programma che gli utenti utilizzano per installare o cambiare le loro password può anche avvertirli quando venga scelta una password troppo facile e tra le altre caratteristiche, si potrebbe lamentare perché:

1. Le password dovrebbero essere lunghe almeno sette caratteri.
2. Le password dovrebbero contenere sia lettere maiuscole sia lettere minuscole.
3. Le password dovrebbero contenere almeno una cifra o un carattere speciale.
4. Le password non dovrebbero essere parole del dizionario, nomi di persona, ecc. .

Un programma per le password clemente potrebbe avere da ridire, mentre uno più rigido potrebbe rifiutare la password e chiederne una migliore; il programma per le password potrebbe anche dare un suggerimento, come discusso in precedenza.

Alcuni sistemi operativi richiedono all'utente di modificare regolarmente la loro password, in modo da limitare i danni nel caso in cui una password venga scoperta: in questo caso accade che se gli utenti devono cambiare troppo spesso la loro password, finiranno con lo scegliere password troppo facili. Se poi si evita che scelgano password facili, per evitare di dimenticarle, le scriveranno su fogli volanti, attaccati ai loro monitor e ciò diventa una falla nella sicurezza ancora più grande!

## Password usa e getta

La forma più radicale per cambiare le password tutte le volte è la password *usa e getta*: quando vengono utilizzate le password *use e getta*, l'utente ha un libro contenente una lista di password e ad ogni accesso utilizza la parola successiva nella lista. Se un intruso scopre una password, non gli può arrecare nessun vantaggio, perché la volta successiva dovrà utilizzare una nuova password. Naturalmente è consigliabile evitare di perdere il libro con le password!

A dire il vero, non c'è bisogno di un libro, per via di uno schema elegante, concepito da Leslie Lamport, che consente ad un utente di eseguire un login sicuro su una rete non sicura, utilizzando password *usa e getta* (Lamport, 1981). Il metodo di Lamport può essere utilizzato per permettere ad un utente che lavori su un home computer di accedere ad un server di Internet, anche se degli intrusi possono vedere e copiare tutto il traffico in entrambe le direzioni. Inoltre non devono essere memorizzate informazioni segrete nel file system né del server né dell'utente del PC.

L'algoritmo è basato su una funzione unidirezionale, cioè una funzione  $y = f(x)$  per la quale data  $x$  è facile trovare  $y$ , ma data  $y$  è praticamente impossibile determinare  $x$ . Inoltre l'input e l'output dovrebbero avere la stessa lunghezza, ad esempio 128 bit.

L'utente sceglie una password segreta che ricorda, ed un intero  $n$ , che rappresenta il

numero di password usa e getta che l'algoritmo è in grado di generare. Consideriamo ad esempio  $n = 4$ , anche se in realtà dovrebbe essere utilizzato un valore di  $n$  molto più grande. Se la password scelta è  $s$ , la prima password usa e getta è ottenuta eseguendo la funzione unidirezionale  $n$  volte:

$$P_1 = f(f(f(f(s))))$$

La seconda password è ottenuta eseguendo la funzione unidirezionale  $n - 1$  volte:

$$P_2 = f(f(f(f(s))))$$

La terza password esegue  $f$  due volte, mentre la quarta password la esegue una volta sola. In generale vale quindi  $P_{i+1} = f(P_i)$ . L'aspetto fondamentale che bisogna sottolineare è che data qualsiasi password nella sequenza, è facile calcolare quella precedente nella sequenza numerica, ma è impossibile calcolare quella successiva. Ad esempio data  $P_2$  è facile risalire a  $P_1$ , ma è impossibile determinare  $P_3$ .

Il server viene inizializzato con  $P_0$  che è proprio  $f(P_0)$ ; questo valore è memorizzato nel file delle password ed associato al nome dell'utente assieme all'intero 1, per indicare che la successiva password richiesta è  $P_1$ . Quando l'utente vuole eseguire il login per la prima volta, manda il suo nome al server il quale risponde spedendogli l'intero nel file delle password, 1. La macchina dell'utente risponde con  $P_1$ , che può essere calcolata localmente a partire da  $s$ , digitata al momento. Il server calcola quindi  $f(P_1)$  e lo confronta con il valore memorizzato nel file delle password ( $P_0$ ). Se il valore corrisponde, l'accesso è consentito, l'intero è aumentato a 2 e  $P_0$  viene sovrascritto da  $P_1$  nel file delle password.

All'accesso successivo, il server manda all'utente un 2, e la macchina dell'utente calcola  $P_2$ ; il server calcola quindi  $f(P_2)$  e lo confronta con quanto memorizzato nel file delle password. Se il valore corrisponde, l'accesso è consentito, l'intero è aumentato a 3 e  $P_1$  viene sovrascritto da  $P_2$  nel file delle password. La proprietà che fa funzionare questo schema è che se un intruso riuscisse anche a carpire  $P_1$ , non sarebbe in grado di calcolare  $P_{i+1}$  a partire da questo, ma solo  $P_{i-1}$ , il quale, essendo già stato utilizzato risulta dunque inutile. Quando tutte le  $n$  password sono state utilizzate, il server viene reinizializzato con una nuova chiave segreta.

## Autenticazione challenge-response

Una variante all'idea di password è quella di fornire ad ogni nuovo utente una lunga sequenza di domande e risposte che vengono poi memorizzate nel server in maniera sicura (ad esempio in forma criptata). Le domande dovrebbero essere scelte in modo che l'utente non abbia bisogno di scriverle. Ecco alcuni esempi:

1. Chi è la sorella di Mariolina?
2. In quale strada si trovava la scuola elementare che hai frequentato?
3. Che cosa insegnava la sig. Brambilla?

Al momento del login, il server pone una di queste a caso e controlla la risposta. Tuttavia per rendere pratico questo schema, occorrebbero molte coppie domanda-risposta.

Una variante è l'autenticazione challenge-response (domanda e risposta): quando viene utilizzata, l'utente sceglie un algoritmo al momento in cui si registra come utente, ad esempio  $x^2$ . Quando l'utente effettua il login, il server gli manda un argomento, diciamo 7, e l'utente in questo caso deve scrivere 49. L'algoritmo inoltre può cambiare dal mattino al pomeriggio, a seconda dei giorni della settimana e così via.

Se il terminale dell'utente ha una vera capacità di calcolo, come un personal computer, un palmare, od un telefono cellulare, è possibile utilizzare anche una forma più elaborata di challenge-response. In precedenza l'utente seleziona una chiave segreta,  $k$ , che è inizialmente portata al server a mano; una copia di questa viene anche memorizzata (in maniera sicura) sul computer dell'utente. Al momento del login, il server manda un numero casuale  $r$  al computer dell'utente, che quindi calcola  $f(r, k)$ , dove  $f$  è una funzione nota pubblicamente, e glielo rimanda indietro. Il server esegue il medesimo calcolo e controlla se il risultato pervenutogli corrisponde. Il vantaggio di questo schema rispetto ad una password è che anche se qualcuno vede e registra tutto il traffico in entrambe le direzioni, non saprà nulla che lo potrà aiutare la prossima volta. Naturalmente la funzione  $f$  deve essere abbastanza complicata, in modo che  $k$  non possa essere dedotto neanche con un ampio numero di osservazioni.

### 9.3.2 Autenticazione tramite oggetti fisici

Il secondo metodo per l'autenticazione degli utenti è controllare alcuni oggetti fisici che questi possiedono, piuttosto che qualcosa che conoscono; le chiavi metalliche delle porte sono state utilizzate per secoli a tale scopo. Oggigiorno invece, l'oggetto fisico utilizzato spesso è una carta di plastica che viene inserita in un lettore collegato ad un terminale o ad un computer. Di solito l'utente deve non solo inserire la carta, ma anche digitare una password, in modo da evitare che qualcuno possa utilizzare una carta rubata o persa. In tale maniera si può utilizzare un bancomat, che permette all'utente di eseguire un login nel computer della banca tramite un terminale remoto (il bancomat) usando una carta di plastica ed una password (attualmente tale password consiste in un codice di 4 cifre, PIN, nella maggior parte dei paesi, ma è solo un modo per evitare la spesa del dover mettere una vera tastiera sul bancomat).

Le informazioni riportate nelle carte di plastica vengono memorizzate in due modi: banda magnetica e chip. Le carte con banda magnetica memorizzano circa 140 byte di informazioni che vengono scritte su un pezzo di nastro magnetico incollato sul retro della carta; queste informazioni possono essere lette da un terminale e mandate al computer centrale. Spesso le informazioni contengono la password dell'utente (ad esempio il codice PIN), in modo che il terminale possa controllare l'identità dell'utente anche se il collegamento con il computer centrale non è attivo; inoltre di solito la password è criptata con una chiave nota solo alla banca. Queste carte costano dai 10 a 50 centesimi (di dollaro) ciascuna, a seconda della presenza o meno di un ologramma adesivo sul fronte di essa e del volume di produzione. In generale l'identificare utenti tramite carte a banda magnetica è rischioso in quanto gli strumenti necessari a leggerle e scriverle sono economici e molto diffusi.

Le carte con chip contengono un circuito integrato (chip) e possono essere ulteriormente suddivise in due categorie: carte a valore immagazzinato e smart card. Le carte a valore immagazzinato hanno poca memoria (di solito meno di 1 KB) ed usano la tecnologia delle EEPROM per memorizzarne il valore quando la carta viene rimossa dal lettore eliminando l'alimentazione. Non essendoci nessuna CPU sulla carta, il valore memorizzato deve essere modificato tramite una CPU esterna (nel lettore). Queste carte sono prodotte a milioni al costo di circa 1 dollaro e sono utilizzate, ad esempio, come carte telefoniche prepagate. Quando viene effettuata una telefonata, il telefono decrementa il valore della carta, senza che vi sia alcun passaggio di denaro. Per questo motivo tali carte vengono generalmente emesse da un'azienda per l'utilizzo nei suoi apparecchi (ad esempio telefoni o distributori automatici). Tali carte potrebbero essere utilizzate anche per l'autenticazione del login, memorizzandovi una password di 1 KB che il lettore manderebbe poi ad un computer centrale, ma ciò accade raramente.

Tuttavia oggi gran parte del lavoro riguardante la sicurezza si è focalizzato sulle **smart card** che attualmente hanno una CPU a 4 Mhz e 8 bit, 16 Kb di ROM, 4 Kb di EEPROM, 512 byte di RAM scratch e un canale di comunicazione col lettore a 9600 bps. Le carte diventano più intelligenti col tempo, ma sono vincolate in diversi modi, tra cui la profondità del chip (perché è inserito nella carta), la larghezza del chip (non si deve rompere quando l'utente piega la carta) e il prezzo (tipicamente da 5 a 50 dollari, a seconda della potenza della CPU, della dimensione della memoria e della presenza o meno di un coprocessore crittografico).

Le smart card possono essere utilizzate per contenere denaro, come accade per le carte a valore immagazzinato, ma con una sicurezza ed una universalità decisamente migliori. Le carte possono essere caricate col denaro ad un bancomat o a casa tramite il telefono ed uno speciale lettore fornito dalla banca; una volta passato attraverso il lettore di un commerciante, l'utente può autorizzare la carta a dedurre una certa quantità di denaro dalla carta stessa (digitando YES), in modo che la carta mandi un breve messaggio criptato al commerciante; il commerciante in seguito può girare il messaggio alla banca per l'accreditamento della somma pagata.

Il grande vantaggio delle smart card rispetto, diciamo, alle carte di credito o debito, è che non hanno bisogno di una connessione attiva con una banca. Se non credete che questo sia un vantaggio, provate il seguente esperimento: cercate di comprare una sola confezione di caramelle in un negozio ed insistete nel voler pagare con una carta di credito; se il negoziante si rifiuta, sostenete di non avere denaro contante con voi e che inoltre avete bisogno di usarla perché avete attivato un programma tipo frequent flyer. Scoprirete che il negoziante non è entusiasta dell'idea (perché il costo associato riduce il profitto sull'articolo). Questo rende perciò le smart card utili per piccoli acquisti, parcheggi a pagamento, telefoni pubblici, distributori automatici, e molti altri congegni che normalmente richiedono l'uso di monete. Queste carte vengono diffusamente utilizzate in Europa e si stanno diffondendo anche altrove.

Le smart card hanno potenzialmente molti altri usi (ad esempio codificare le allergie del proprietario, ed altre informazioni mediche in modo sicuro per poterle utilizzare in caso di emergenza), ma non è il posto adatto per raccontare tali storie, perché il nostro interesse è come possono essere utilizzate per effettuare un login sicuro. Il concetto base è semplice: una smart card è un piccolo computer a prova di manomissione che può stabilire una connessione (chiamata protocollo) con un computer centrale per autenticare l'utente. Ad esempio un utente che voglia acquistare qualcosa in un sito Web di e-commerce potrebbe inserire una smart card in un lettore situato in casa sua e collegato al suo PC. Quindi il sito di e-commerce utilizzerebbe la smart card non solo per autenticare l'utente in maniera più sicura rispetto ad una password, ma potrebbe anche dedurre il costo dell'acquisto direttamente dalla smart card, eliminando molti problemi (e rischi) associati all'uso di una carta di credito per gli acquisti online.

Possono essere utilizzati vari schemi di autenticazione con una smart card; un semplice schema challenge-response lavora così: il server manda un numero casuale di 512 bit alla smart card, che vi aggiunge la password di 512 bit memorizzata nella EEPROM della carta; la somma viene quindi elevata al quadrato ed i 512 bit centrali vengono rimandati al server che conosce la password dell'utente e può quindi calcolare se il risultato sia corretto o meno. Tale sequenza di operazioni viene mostrata in Figura 9.7. Se qualcuno riuscisse anche ad intercettare entrambi i messaggi, non sarebbe in grado di fare nulla di sensato con essi, ed anche il registrarli per un successivo utilizzo è inutile, in quanto al prossimo login verrà trasmesso un differente numero casuale di 512 bit. Naturalmente si può utilizzare un algoritmo molto più stravagante del quadrato, e di solito lo si fa.

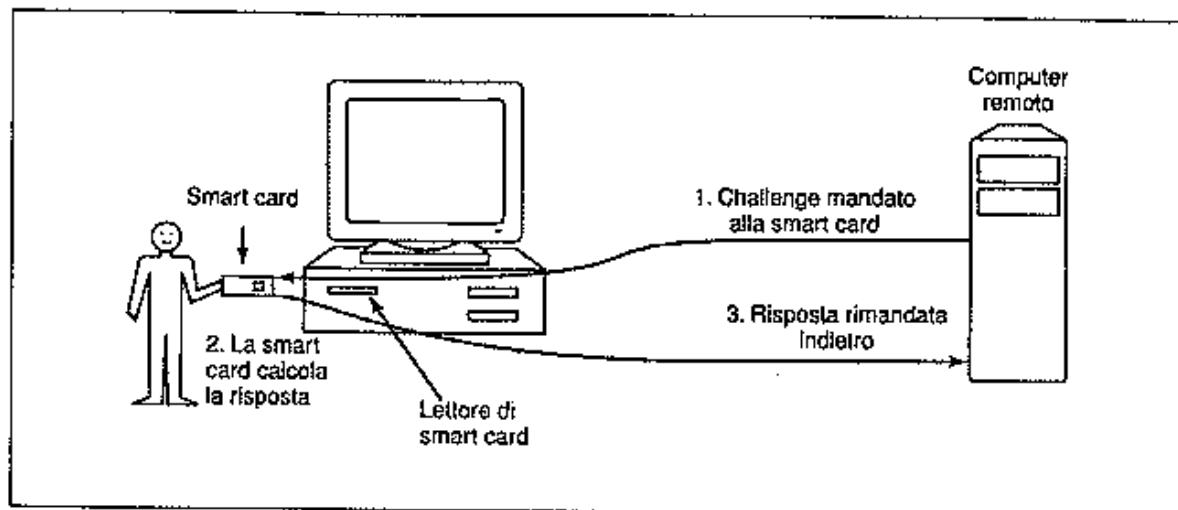


Figura 9.7 Uso di una smart card per l'autenticazione.

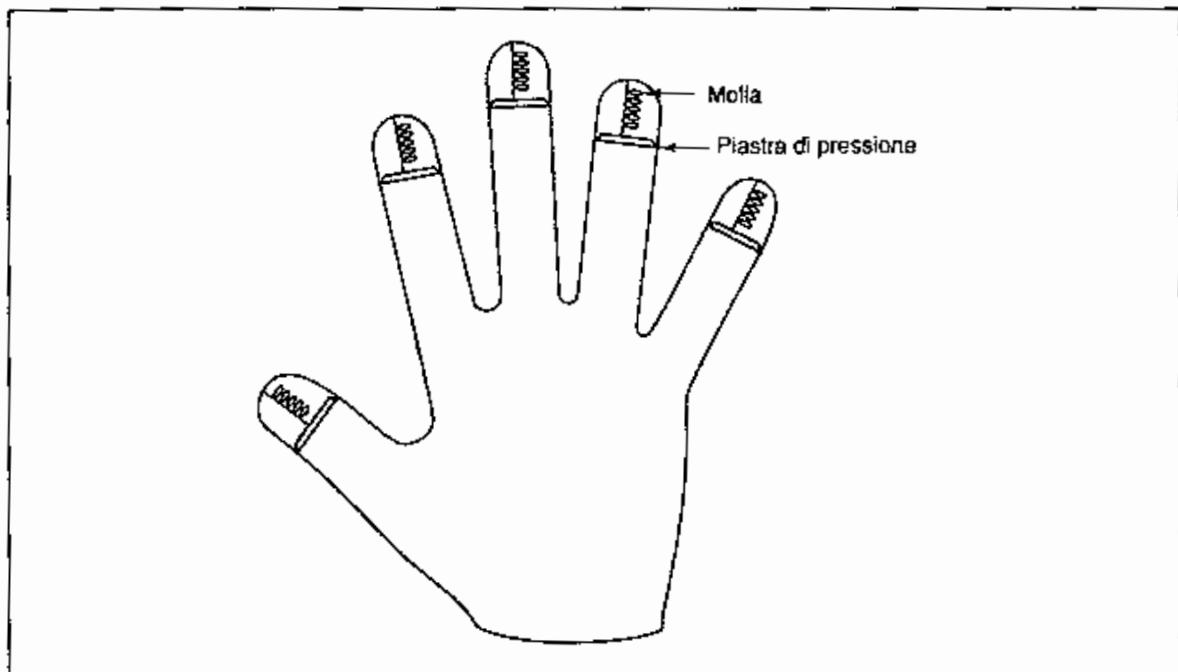
Uno svantaggio di ogni protocollo crittografico fissato è che nel corso del tempo potrebbe essere infranto, rendendo inutile la smart card. Un modo per evitare che questo accada è utilizzare la ROM della carta non per un protocollo crittografico ma per un interprete Java: il vero protocollo crittografico viene poi caricato sulla carta come un programma binario Java e interpretato. In questo modo non appena un protocollo viene infranto, se ne può installare uno nuovo ovunque all'istante. Uno svantaggio di questo approccio è che rende ancora più lenta una carta già lenta, ma col miglioramento della tecnologia, questo metodo diventa molto flessibile. Un altro svantaggio delle smart card è che una volta persa o smarrita, può essere sottoposta ad un attacco con analisi della corrente: osservando la potenza elettrica consumata durante operazioni di crittazione ripetute, un esperto fornito dell'attrezzatura adeguata è in grado di dedurre la chiave; misurando poi il tempo di crittazione con diverse chiavi scelte appositamente, è anche in grado di ottenere informazioni preziose sulla chiave.

### 9.3.3 Autenticazione tramite biometrie

Il terzo metodo di autenticazione misura le caratteristiche fisiche dell'utente che sono difficilmente falsificabili. Tali caratteristiche sono chiamate **biometrie** (Pankanti et al., 2000); ad esempio un lettore di impronte, digitali o vocali, nel terminale potrebbe verificare l'identità dell'utente.

Il tipico sistema biometrico è composto da due parti: registrazione ed identificazione. Durante la registrazione vengono misurate le caratteristiche dell'utente ed i risultati vengono digitalizzati; quindi le caratteristiche significative vengono estratte e memorizzate in un record associato all'utente. Tale record può essere conservato in un database centrale (ad esempio per effettuare il login su un computer remoto), oppure memorizzato in una smart card che l'utente può portarsi in giro ed inserire in un lettore remoto (ad esempio un bancomat).

La seconda parte è l'identificazione: l'utente si rende visibile e fornisce la sua account; il sistema esegue quindi delle nuove misurazioni. Se i nuovi valori corrispondono a quelli campionati al momento della registrazione, il login è accettato, altrimenti rifiutato. L'account è necessaria in quanto le misure non sono esatte, pertanto è difficile assegnare loro un indice ed effettuare delle ricerche sugli indici; inoltre due persone possono avere le medesime caratteristiche fisiche, perciò il richiedere che le caratteristiche misurate corri-



**Figura 9.8** Un dispositivo per misurare la lunghezza delle dita.

spondano a quelle di uno specifico utente è più restrittivo che richiedere che corrispondano a quelle di un qualsiasi utente.

Le caratteristiche scelte dovrebbero poter variare in un ampio intervallo in modo che il sistema possa distinguere molte persone senza errore; ad esempio il colore dei capelli non è un buon discriminante in quanto troppe persone hanno lo stesso colore. Oltre a ciò, le caratteristiche scelte non dovrebbero variare molto nel tempo; ad esempio la voce di una persona può essere differente a causa di un raffreddore, ed un viso può apparire diverso per via della barba o del trucco non presenti al momento della registrazione. Siccome i campioni successivi non corrisponderanno mai esattamente ai valori presi al momento della registrazione, i progettisti del sistema devono decidere quanto deve essere precisa la corrispondenza per essere accettata; in particolare devono decidere se sia peggio rifiutare il login ad un legittimo utente una volta ogni tanto, oppure permettere ad un impostore di accedere una volta ogni tanto. Un sito specializzato in e-commerce potrebbe decidere che rifiutare un cliente legale potrebbe essere peggio che accettare una minima frode, al contrario un sito di armi nucleari potrebbe decidere che rifiutare l'accesso ad un impiegato autentico sia meglio che permettere l'accesso ad un qualsiasi sconosciuto due volte l'anno.

Diamo ora una rapida occhiata alle biometrie attualmente in uso. L'analisi della lunghezza delle dita è sorprendentemente pratica: quando questa tecnica viene utilizzata, ogni terminale è dotato di un dispositivo simile a quello di Figura 9.8. L'utente vi infila la sua mano e la lunghezza di tutte le sue dita viene misurata e verificata all'interno del database.

Tuttavia la misurazione della lunghezza delle dita non è perfetta: il sistema può essere attaccato con stampi a forma di mano fatti di gesso o di qualche altro materiale, magari con dita modificabili per permettere varie prove.

Un'altra biometria che sta guadagnando in popolarità è l'analisi del modello retinico: ognuno ha una diversa conformazione dei vasi sanguigni della retina, anche i gemelli identici; tale conformazione può essere accuratamente fotografata da una macchina foto-

grafica posta ad un metro dal soggetto, senza che la persona ne sia nemmeno a conoscenza. La quantità di informazioni contenute in una analisi retinica è molto maggiore di quella fornita da un'impronta digitale, e può essere codificata in circa 256 byte.

Qualsiasi tecnica che si basi su immagini è soggetta allo spoofing; ad esempio una persona potrebbe avvicinarsi alla macchina fotografica di un bancomat indossando occhiali scuri ai quali siano state attaccate le foto delle retine di qualcun altro. Dopotutto, se la macchina fotografica di un bancomat può scattare buone foto dalla distanza di un metro, anche altre persone possono farlo, anche a distanze maggiori utilizzando dei teleobiettivi. Per questa ragione di solito si usano videocamere e non macchine fotografiche, e si cercano le pulsazioni normalmente presenti nei vasi sanguigni delle retine.

Una tecnica in un certo modo differente è l'analisi della firma: l'utente scrive il suo nome con una penna speciale collegata ad un terminale, ed il computer la confronta con un campione memorizzato online o su una smart card. Ancora meglio se si confronta non la firma, ma i movimenti e la pressione esercitata mentre si firma; un buon falsoario può essere in grado di falsificare una firma, ma non avrà idea dell'ordine esatto in cui i tratti sono stati tracciati, o con che velocità e con quale pressione siano stati eseguiti.

Uno schema che si basa poco su hardware speciale è la biometria vocale (Markovitz, 2000): tutto ciò di cui si ha bisogno è un microfono (o anche un telefono); il resto è software. A differenza dei sistemi a riconoscimento vocale, che cercano di capire quello che sta dicendo chi parla, questi sistemi cercano di determinare chi stia parlando. Alcuni sistemi richiedono all'utente di dire una password segreta, ma tali sistemi possono essere aggirati da un "origliatore" che può registrare le password e duplicarle successivamente. Sistemi più avanzati dicono qualcosa all'utente e chiedono di ripeterla, utilizzando testi differenti ad ogni login. Alcune aziende stanno iniziando ad utilizzare l'identificazione vocale per applicazioni come l'home shopping tramite telefono, perché l'identificazione vocale è meno soggetta a frode rispetto all'utilizzo di un PIN per l'identificazione.

Potremmo andare avanti ancora a lungo con altri esempi, ma questi ultimi due sottolineeranno un aspetto importante. I gatti e molti altri animali per marcire il loro territorio, orinano sul suo perimetro; apparentemente i gatti possono identificarsi l'un l'altro in questa maniera. Supponiamo che qualcuno salti fuori con un piccolo dispositivo in grado di effettuare istantaneamente le analisi delle urine, fornendo quindi un'identificazione certissima. Ogni terminale potrebbe essere dotato di uno di questi dispositivi, con una discreta scritta "Per effettuare la login, inserire qui il campione": questo potrebbe essere un sistema assolutamente impenetrabile, tuttavia vi sarebbe seri problemi a farlo accettare da parte degli utenti.

Lo stesso si potrebbe dire di un sistema formato da una puntina da disegno ed un piccolo spettrografo: l'utente dovrebbe premere il suo pollice contro la puntina, estraendo in questo modo una goccia di sangue per l'analisi spettrografica. Il punto di tutto ciò è che uno schema di autenticazione deve essere psicologicamente accettabile dalla comunità degli utenti; la misurazione della lunghezza della dita probabilmente non causerà alcun problema, ma anche qualcosa di non intrusivo come memorizzare le impronte digitali online può risultare inaccettabile a molti, in quanto le impronte digitali possono essere associate ai criminali.

### 9.3.4 Contromisure

I centri di calcolo che prendono sul serio il problema della sicurezza, cosa che accade di frequente il giorno dopo che un intruso è penetrato ed ha provocato gravi danni, spesso prendono provvedimenti per rendere molto più difficile l'accesso non autorizzato. Ad

esempio un'azienda potrebbe adottare una politica che permetta l'accesso ai terminali a chi lavora nel dipartimento brevetti solo tra le 8 di mattina e le 5 del pomeriggio, dal lunedì al venerdì e solamente da una macchina del dipartimento brevetti collegata alla rete LAN dell'azienda: qualsiasi tentativo di effettuare un login da parte di un impiegato del dipartimento brevetti all'ora o dalla postazione sbagliata sarebbe considerato come un tentativo di violazione.

Anche la chiamata da linea telefonica può essere resa sicura come segue: chiunque può chiamare ed effettuare il login, ma dopo che il login ha avuto successo, interrompe immediatamente la connessione e richiama l'utente ad un numero concordato precedentemente. Questo metodo implica che un intruso non può semplicemente cercare di penetrare il sistema da un qualsiasi telefono; solo il telefono (di casa) dell'utente ci riuscirà. In ogni caso, con o senza la chiamata di conferma, il sistema dovrebbe richiedere almeno 5 secondi per verificare ogni password digitata dall'utente collegato tramite linea telefonica, e dovrebbe incrementare questo tempo dopo diversi tentativi infruttuosi di fare login, al fine di ridurre il numero di tentativi che un intruso ha a disposizione. Dopo tre tentativi falliti, la linea dovrebbe essere staccata per 10 minuti e si dovrebbe segnalare il fatto al personale addetto alla sicurezza.

Tutti i login dovrebbero essere registrati: quando un utente effettua il login, il sistema dovrebbe comunicare l'ora ed il terminale dell'ultimo login, in modo che l'utente possa rilevare una possibile violazione.

Il passo successivo è quello di collocare trappole con esche per catturare intrusi: un semplice schema consiste nell'utilizzare un account speciale che abbia una password facile (ad esempio login: guest, password: guest); ogni volta che qualcuno effettua un login utilizzando questo account, gli addetti alla sicurezza del sistema vengono immediatamente avvisati. Tutti i comandi digitati dall'intruso vengono immediatamente visualizzati sul monitor del capo della servizio di sicurezza in modo che questi possa vedere esattamente cosa sta facendo l'intruso.

Altre trappole possono essere bachi del sistema operativo facile da trovare, o cose del genere, progettate comunque col proposito di catturare gli intrusi nel momento in cui agiscono. Stoll (1989) ha scritto un'una divertente descrizione della trappole che aveva sistematizzato per scovare una spia che si era introdotta in un computer dell'università alla ricerca di segreti militari.

## 9.4 Attacchi dall'interno del sistema

Una volta che un cracker sia riuscito a effettuare il login in un computer, può cominciare a fare danni. Se il computer ha buon sistema di sicurezza, sarà possibile solamente causare danni all'utente la cui account è stata violata, ma spesso questo ingresso iniziale può servire come punto di partenza per violare altre account in seguito. Nelle sezioni seguenti esamineremo attacchi che possono essere mossi da qualcuno già all'interno del sistema, sia un cracker che è penetrato illecitamente, sia, talvolta, un utente legittimo ma con del rancore verso qualcuno.

### 9.4.1 Cavalli di Troia

Un antico attacco proveniente dall'interno è il cavallo di Troia, nel quale un programma apparentemente innocente contiene codice per eseguire azioni inaspettate ed indesiderabili. Queste operazioni possono riguardare la modifica, la cancellazione o la

crittazione di file dell'utente, copiandoli laddove il cracker potrà successivamente recuperarli, o anche mandandoli al cracker stesso o nascondendoli in aree sicure tramite e-mail o FTP. Per fare in modo che il cavallo di Troia agisca, la persona che lo ha progettato deve fare in modo che il programma che lo trasporta venga eseguito. Un modo per realizzare questo è mettere il programma a disposizione su Internet, come un nuovo eccitante gioco gratuito, un lettore di Mp3, un lettore porno "speciale", o qualcosa del genere, cercando possibilmente di attrarre l'attenzione ed incoraggiare la gente a scaricarlo. Quando questo va in esecuzione, viene chiamata la procedura del cavallo di Troia, che può fare qualsiasi cosa sia consentita all'utente (ad esempio cancellare file, attivare una connessione, eccetera). Notiamo che questo stratagemma non richiede all'autore del cavallo di Troia di violare il computer della vittima.

Ci sono altri metodi per ingannare la vittima e fare eseguire anche un cavallo di Troia; ad esempio molti utenti UNIX hanno una variabile d'ambiente, \$PATH, che stabilisce in quali directory si debba cercare un comando. Tale variabile si può visualizzare digitando in una shell il comando

```
echo $PATH
```

Una possibile impostazione per l'utente *ast* su un particolare sistema potrebbe contenere le seguenti directory:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Naturalmente altri utenti possono avere un path di ricerca diverso. Quando l'utente digita

```
prog
```

nella shell, questa dapprima controlla se vi sia un programma con nome */usr/ast/bin/prog*; se c'è, lo esegue; se non c'è, la shell cerca */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog* e così via, tentando tutte e 11 le directory a turno prima di smettere. Supponiamo che una di queste directory sia stata lasciata non protetta, in questo modo il cracker potrebbe mettere lì un programma. Se questa è la prima occorrenza del programma nella lista, verrà eseguita ed assieme a questo verrà eseguito anche il cavallo di Troia.

La maggior parte dei programmi più comuni si trovano in */bin* o */usr/bin*, perciò mettere il cavallo di Troia in */usr/bin/X11/ls* non funzionerà solitamente in quanto il programma *ls* vero verrà trovato per primo. Nonostante ciò supponiamo che il cracker inserisca *la* in */usr/bin/X11*; se sbagliando, un utente scrive *la* invece di *ls* (il comando che restituisce un listato dei contenuti della directory), ora il cavallo di Troia verrà eseguito, compiendo il suo sporco lavoro e poi restituendo il messaggio (corretto) che *la* non esiste. Inserendo cavalli di Troia in complicate directory che difficilmente si vanno ad esaminare, e dando ad essi dei nomi che possono rappresentare comuni errori di battitura, c'è una buona possibilità che presto o tardi qualcuno chiami uno di questi; e nel caso in cui quel qualcuno sia un superuser anche i superuser fanno errori di battitura!, il cavallo di Troia avrà la possibilità di sostituire */bin/ls* con una versione contenente il cavallo di Troia, che sarà quindi chiamato ogni volta da quel momento in avanti.

Un utente malvagio ma legale, Mal, potrebbe anche sistemare una trappola per il superuser come segue: costui mette nella sua home directory una versione di *ls* contenente un cavallo di Troia, e poi compie qualcosa di sospetto per ottenere di sicuro l'attenzione del

superuser, ad esempio lanciando 100 processi di calcolo intensivo tutti assieme. Probabilmente il superuser li controllerà digitando

```
cd /usr/mal
ls -l
```

per vedere quello che c'è nella home directory di Mal. Visto che alcune shell cercano prima nelle directory locali e poi iniziano a lavorare con \$PATH, il superuser può avere chiamato il cavallo di Troia di Mal con diritti da superuser. A questo punto il cavallo di Troia potrebbe attribuire a /usr/mal/bin/sh il SETUID di root; tutto ciò di cui ha bisogno sono due chiamate di sistema: chown per cambiare il proprietario di /usr/mal/bin/sh a root e chmod per modificare il bit del suo SETUID. In questo modo Mal può diventare superuser a suo piacimento eseguendo tale shell.

Se Mal si trova spesso a corto di denaro, potrebbe utilizzare uno dei cavalli di Troia fraudolenti seguenti per migliorare la sua posizione finanziaria. Nel primo caso, il cavallo di Troia controlla se la vittima ha un programma di banking online installato, ad esempio Quicken; se è così, il cavallo di Troia istruisce tale programma affinché parte del denaro della vittima venga trasferito su un conto fittizio (preferibilmente in un paese lontano), in modo da poterlo recuperare successivamente sotto forma di denaro liquido.

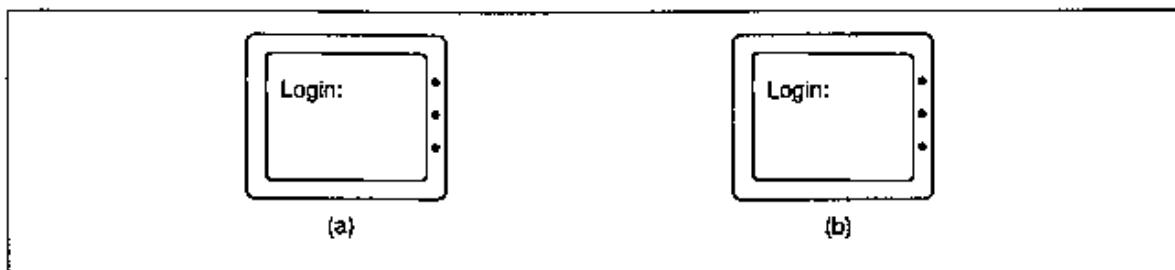
Nel secondo caso, descritto supponendo che Mal sia negli Stati Uniti, il cavallo di Troia dapprima disattiva il suono del modem, quindi compone un numero 900 (a pagamento), ancora una volta preferibilmente in un paese lontano, quale la Moldavia (una parte dell'ex Unione Sovietica).

Se l'utente era on-line quando il cavallo di Troia ha iniziato ad agire, allora il 900 deve essere un Internet provider (inoltre costoso), di modo che l'utente non si accorga di nulla e possa stare on-line per ore. Nessuna di queste tecniche è ipotetica: entrambe sono state utilizzate e riportate (Denning, 1999). Nella seconda furono accumulati 800000 minuti di connessione in Moldavia prima che la Commissione Federale del Commercio degli Stati Uniti riuscisse a staccare la spina e facesse causa a tre persone di Long Island. Alla fine costoro decisero di restituire 2,74 milioni di dollari a 38000 vittime.

## 9.4.2 Login spoofing

Il login spoofing è in qualche modo collegato ai cavalli di Troia e funziona come segue. Di solito quando nessuno è collegato ad un terminale o ad una workstation o a una rete LAN UNIX, appare uno schermo come quello mostrato in Figura 9.9(a). Quando un utente si siede e digita la sua account, il sistema richiede una password; se questa è corretta, l'utente può accedere e partire una shell.

Consideriamo ora questo scenario: Mal scrive un programma che mostri lo schermo della Figura 9.9(b); sembra sorprendentemente uguale allo schermo della Figura 9.9(a), tranne per il fatto che non è in esecuzione il programma di login del sistema, ma bensì uno fasullo scritto da Mal. Mal a questo punto si allontana e guarda quello che accade da una distanza di sicurezza. Quando un utente si siede e digita la sua account, il programma risponde chiederlo una password e disabilitando la visualizzazione sullo schermo (echo); dopo che il login e la password sono state catturate, vengono scritte in un file ed il programma di login fasullo manda un segnale per ammazzare la sua shell. Questa azione permette a Mal di uscire e dà l'avvio al vero programma di login, facendo comparire sullo schermo il prompt di Figura 9.9(a). L'utente pensa di avere commesso un errore di battitura e cerca di accedere nuovamente; questo collegamento ha successo, ma nel frattempo Mal ha ottenuto un'altra coppia (login, password). In questo modo, collegandosi



**Figura 9.9** (a) Schermata di login corretta. (b) Schermata di login falsa.

a molti computer e lanciando su tutti il programma per lo spoofing delle login. Mal può catturare molte password.

L'unico vero modo per proteggersi da ciò è fare in modo che l'accesso avvenga solo dopo una combinazione di tasti che i programmi degli utenti non possano catturare. Windows 2000 utilizza a questo scopo la combinazione CTRL-ALT-DEL: se un utente siende ad un terminale e inizia a digitare CTRL-ALT-DEL, l'utente corrente viene sconnesso e comincia il programma di login. Non c'è modo di oltrepassare questo meccanismo.

### 9.4.3 Bombe logiche

Un altro attacco dall'interno, in questi tempi di grande mobilità degli impiegati è la **bomba logica**. Questo congegno è un pezzo di codice scritto da uno dei programmatore dell'azienda, attualmente dipendente, ed inserito segretamente nel sistema operativo attualmente in uso. Finché il programmatore gli fornisce quotidianamente la sua password, non fa niente; tuttavia se il programmatore è improvvisamente licenziato e fisicamente rimosso dal suo posto senza avvertimenti, il giorno dopo (o la settimana successiva) la bomba logica non ottenendo la sua password quotidiana, scoppierebbe. Sono possibili molte varianti di questo schema: in un caso famoso la bomba logica controllava l'organico; se il numero di codice personale del programmatore non vi appartiva per due giornate consecutive, esplodeva (Spafford et al., 1989).

"Esplodere" potrebbe significare cancellare i dischi, cancellare file a caso, effettuare cambiamenti ai programmi chiave difficilmente rilevabili, o crittare file vitali. Nell'ultimo caso, l'azienda deve compiere una difficile scelta tra il chiamare la polizia (il che può o meno sfociare in una condanna diversi mesi più tardi, ma non recupererà i file perduti) oppure cedere a questo ricatto e ri-assumere l'ex programmatore nelle veci di "consulente", pagando una cifra astronomica per risolvere il problema (e sperando che costui non progetti altre bombe logiche mentre fa ciò).

### 9.4.4 Porte nascoste

Un'altra falla della sicurezza causata dall'interno è la **porta nascosta**. Questo problema è dovuto a codice inserito nel sistema da un programmatore di sistema per oltrepassare alcuni controlli normali. Ad esempio un programmatore potrebbe aggiungere del codice al programma di login per permettere a chiunque di ottenere l'accesso utilizzando la account "zzzzz", senza tenere conto di quanto contiene il file delle password. Il codice normale del programma di login potrebbe essere come quello mostrato in Figura 9.10(a); la porta nascosta lo cambia in quello di Figura 9.10(b). Quello che fa la chiamata a `strcmp` è vedere se l'account è "zzzzz"; se è così, si riesce ad accedere qualunque password venga digitata. Se questo codice di porta nascosta è stato inserito da un program-

```

while (TRUE) {
 printf("login: ");
 prendi_stringa(nome);
 disabilita_eco();
 printf("password: ");
 prendi_stringa(password);
 abilita_eco();
 v = controlla(nome, password);
 if (v) break;
}
esegui_shell(nome);

```

(a)

```

while (TRUE) {
 printf("login: ");
 prendi_stringa(nome);
 disabilita_eco();
 printf("password: ");
 prendi_stringa(password);
 abilita_eco();
 v = controlla(nome, password);
 if (v || strcmp(nome, "zzzzz") == 0) break;
}
esegui_shell(nome);

```

(b)

**Figura 9.10** (a) Codice normale, (b) Codice con inserita un porta nascosta.

matore che lavora per un'azienda informatica, e poi caricato sui suoi computer, il programmatore potrebbe penetrare in ogni computer fabbricato dalla sua azienda, indipendentemente dal proprietario e dal file delle password. La porta nascosta oltrepassa semplicemente tutto il processo di autenticazione.

Un modo in cui le aziende possono prevenire l'inserimento di trabocchetti è utilizzare la **verifica del codice** come pratica standard; tramite questa tecnica, una volta che il programmatore abbia finito di scrivere e testare un modulo, tale modulo viene controllato in un database di codice. Periodicamente tutti i programmatori di un gruppo si riuniscono ed ognuno a turno spiega al resto del gruppo il funzionamento del suo codice, riga per riga: questo non solo accresce enormemente la possibilità che una porta nascosta venga rilevata, ma aumenta la posta in gioco per il programmatore, in quanto essere colto in fallo probabilmente non è un buon viatico per la carriera. Se i programmatori protestano troppo quando tale idea è proposta, una ulteriore possibilità è che due collaboratori testimonino reciprocamente il loro codice.

## 9.4.5 Buffer overflow

Una notevole fonte di attacchi è dovuta al fatto che virtualmente tutti i sistemi operativi e la maggior parte dei programmi di sistema sono scritti in C (perché piace ai programmatori e può essere compilato in maniera estremamente efficiente). Sfortunatamente nessun compilatore C controlla i limiti degli array; di conseguenza anche il seguente pezzo di codice illegale non viene controllato:

```

int i;
char c[1024];
i = 12000;
c[i] = 0;

```

Il risultato è che alcuni byte di memoria (10976 byte al di fuori dell'array *c*) sono sovrascritti, probabilmente con conseguenze disastrose, e non viene eseguito alcun controllo a tempo di esecuzione per prevenire questo errore.

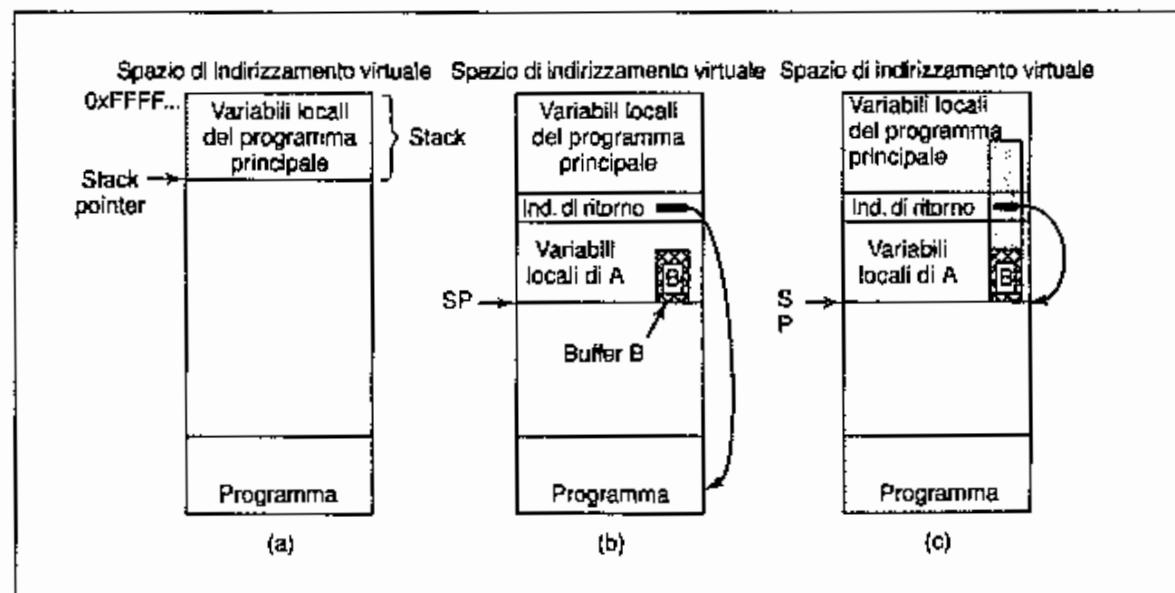
Questa proprietà del C porta ad attacchi nel modo seguente: in Figura 9.11(a) vedia-

mo il programma principale in esecuzione, con le sue variabili locali nello stack; ad un certo punto, questo chiama una procedura *A*, come mostrato in Figura 9.11(b). La normale sequenza della chiamata comincia mettendo sullo stack l'indirizzo di ritorno (che punta all'istruzione successiva alla chiamata); il controllo poi viene trasferito da questo ad *A*, che decrementa lo stack pointer in modo da allocare memoria per le sue variabili locali.

Supponiamo che il lavoro di *A* richieda l'acquisizione del path completo (magari concatenando il path della directory corrente con il nome del file), aprendolo o eseguendo qualche altra operazione su di esso. *A* ha un buffer di dimensioni fissate, (cioè un array) *B*, per memorizzare il nome del file, come mostrato in Figura 9.11(b). Utilizzare un buffer di dimensione fissata per memorizzare il nome del file è di gran lunga più facile da realizzare che determinare la vera dimensione, e poi allocare dinamicamente abbastanza memoria. Se il buffer è di 1024 byte, dovrebbe gestire tutti i nomi di file, vero? Sì, specialmente se il sistema operativo limita i nomi dei file (o, ancora meglio, i path completi) ad un massimo di 255 caratteri.

Sfortunatamente questo modo di ragionare contiene un difetto fatale: supponiamo che l'utente del programma fornisca un nome del file lungo 2000 caratteri; non si potranno aprire file con tale nome, ma chi attacca non se ne preoccupa. Quando la procedura copia il nome del file nel buffer, il nome supera la capacità del buffer e sovrascrive la memoria come mostrato dall'area grigia in Figura 9.11(c), o, ancora peggio, se il nome del file è abbastanza lungo, sovrascrive l'indirizzo di ritorno, così quando *A* ritorna, in realtà prende la parte centrale del nome del file. Se questo indirizzo è casuale, il programma salterà ad un indirizzo casuale e probabilmente andrà in crash entro poche istruzioni.

Ma cosa succede se il nome del file non contiene qualcosa di casuale, ma un programma binario valido, realizzato in maniera talmente attenta che la parola che si sovrappone all'indirizzo di ritorno è proprio l'indirizzo dell'inizio del programma, ad esempio, l'indirizzo di *B*? Ciò che accade è che quando *A* ritorna, il programma, ora in *B*, inizierà ad essere eseguito. In effetti, l'intruso ha inserito il codice nel programma, ed ha ottenuto che esso sia eseguito.



**Figura 9.11** (a) Situazione in cui il programma principale è in esecuzione. (b) Dopo che è stata chiamata la procedura *A*. (c) Buffer overflow mostrato dal colore grigio.

Questo stesso trucco funziona anche con cose differenti dai nomi dei file: funziona con stringhe di ambiente molto lunghe, input dell'utente o qualunque altra cosa dove il programmatore abbia creato un buffer a dimensione fissata per gestire una stringa fornita dall'utente che ci si aspetta sia breve. Fornendo una stringa lunga e fatta "a mano", contenente un programma, è possibile che il programma passi sullo stack e venga eseguito: la funzione `gets` della libreria C, funzione che legge una stringa (di dimensione non nota) in un buffer di dimensione fissata, ma senza controllo per l'overflow, è nota per essere soggetta a questo tipo di attacchi. Alcuni compilatori controllano anche se la funzione `gets` viene utilizzata e avvisano se ciò accade.

Ora arriva la parte peggiore: supponiamo che il programma che è stato attaccato sia il SETUID di root in UNIX (o sia un programma con i diritti di administrator in Windows 2000, che effettivamente è la stessa cosa); il codice inserito ora può fare un paio di chiamate di sistema per far ottenere al file shell sul disco di chi attacca il SETUID di root, in modo che quando venga eseguito, abbia i diritti di superuser. In alternativa ora può mappare una libreria condivisa, appositamente preparata, che può fare danni di ogni genere, o può semplicemente eseguire una chiamata di sistema `exec` e sovrapporsi al programma corrente con la shell, creando una shell con diritti superuser. Una parte notevole di tutti i problemi legati alla sicurezza sono proprio dovuti a questo difetto, che è difficilmente rimediabile, perché esistono troppi programmi C in commercio che non si preoccupano dell'overflow di buffer.

Rilevare che un programma causa overflow di buffer è semplice: passategli nomi di file lunghi 10000 caratteri, stipendi a 100 cifre, o qualcosa di ugualmente inaspettato, e controllate se fallisce generando un core dump. Il passo successivo è quello di analizzare il core dump per vedere dove è stata memorizzata la stringa lunga; da questo, non è affatto difficile capire quale carattere abbia sovrascritto l'indirizzo di ritorno. Se è disponibile il codice sorgente, come accade per la maggior parte dei programmi UNIX, l'attacco è ancora più facile perché la struttura dello stack è nota in precedenza: ci si può difendere da tale attacco inserendo nel codice un controllo esplicito della lunghezza di tutte le stringhe fornite dall'utente, prima di passarle a buffer di lunghezza fissata. Sfortunatamente, il fatto che alcuni programmi siano vulnerabili a questo genere di attacchi, viene fuori solamente dopo un attacco che ha avuto successo.

#### 9.4.6 Generici attacchi alla sicurezza

Il modo normale per verificare la sicurezza di un sistema è raccogliere un gruppo di esperti, noti come **tiger team** o **penetration team**, per vedere se siano capaci di accedervi. Hebbard ed altri (1980) provarono la stessa cosa con un gruppo di studenti. Nel corso degli anni, questi team hanno scoperto un numero di aree in cui i sistemi sono probabilmente deboli. Nel seguito sono elencati gli attacchi che più di frequente hanno successo e, sebbene questi siano stati progettati per attaccare sistemi a timesharing, possono essere anche utilizzati per attaccare server LAN ed altre macchine condivise. Pertanto, quando si progetta un sistema, bisogna assicurarsi che possa resistere ad attacchi come i seguenti:

1. Richiesta di pagine di memoria, spazio su disco o su nastri e loro lettura. Molti sistemi non ne cancellano il contenuto prima di allocarli, e questi possono contenere informazioni interessanti scritte dal precedente proprietario.
2. Tentativi di chiamate di sistema illecite, o chiamate legali, ma con parametri illegali, o anche chiamate di sistema legali con parametri legali, ma non ragionevoli, come nomi di file lunghi migliaia di caratteri. Molti sistemi si possono facilmente confondere.

3. Avviare un login e poi digitare DEL, CANC o BREAK a metà della sequenza di login. In alcuni sistemi il programma che controlla il login terminerà e il login avrà buon esito.
4. Tentativi di modifica delle strutture complesse del sistema operativo mantenute nello spazio utente (se esiste). In molti sistemi (specialmente nei mainframe), per aprire un file, il programma crea una grande struttura dati che contiene il nome del file e molti altri parametri e li passa al sistema. Quando si legge o si scrive il file, il sistema qualche volta aggiorna la struttura stessa. Cambiando questi campi si può causare un guasto nel sistema di sicurezza.
5. Cercare manuali che dicono "non fare X" e provare il maggior numero possibile di variazioni di X!
6. Convincere un programmatore di sistema ad inserire una porta nascosta nel sistema, per evitare che alcuni controlli, vitali per la sicurezza di tutti gli utenti, siano effettuati per una certa account.
7. Se tutto il resto fallisce, allora l'intruso potrebbe cercare la segretaria dell'amministratore di sistema e raggiungerla spacciandosi per un ingenuo utente che ha dimenticato la sua password di cui ha urgente bisogno. Un approccio alternativo è corrompere la segretaria: la segretaria probabilmente ha facile accesso a tutti i tipi di informazioni utili, e di solito non è ben retribuita. Non si sottovalutino i problemi causati dal personale.

Questi ed altri tipi di attacchi sono stati descritti da Linde (1975) e sebbene sia un vecchio articolo, gli attacchi ivi descritti spesso funzionano.

### 9.4.7 Famoso violazioni della sicurezza

Così come l'industria dei trasporti ha i disastri del *Titanic*, dell'*Hindenburg*, e del *Concorde*, gli esperti di sicurezza hanno alcune cose che vorrebbero dimenticare. In questa sezione si studieranno alcuni problemi di sicurezza interessanti che si sono incontrati in tre diversi sistemi operativi: UNIX, TENEX e OS/360.

#### Violazioni della sicurezza famose in UNIX

L'utilità *lpr* di UNIX, che stampa un file su una stampante, ha un'opzione per rimuovere il file dopo che è stato stampato. Nelle prime versioni di UNIX era possibile per chiunque usare *lpr* per stampare il file delle password e poi fare sì che il sistema lo rimuovesse.

Un altro modo per entrare abusivamente in UNIX era collegare un file detto *core* nella directory di lavoro con il file delle password. L'intruso poi forzava un core dump di un programma SETUID, che il sistema scriveva sul file *core*, cioè sopra il file delle password. In questo modo un utente poteva sostituire il file di password con uno che conteneva alcune stringhe di sua scelta (ad esempio, comandi).

Un altro difetto di UNIX aveva a che fare con il comando

```
mkdir foo
```

*mkdir*, che era un programma SETUID posseduto da root, prima creava l'i-node per la directory *foo* con la chiamata di sistema *mknod* e poi cambiava il proprietario di *foo* sostituendo l'UID effettivo (cioè root) con quello dell'utente. Quando il sistema era lento, qualche volta l'utente riusciva a rimuovere velocemente l'i-node della directory e fare un

link al file di password con il nome *foo*, dopo mknod, ma prima di chown. Quando mkdir eseguiva chown, rendeva l'utente proprietario del file di password. Quindi, mettendo i comandi necessari in uno script di shell, si poteva provarli più volte finché il trucco non fosse riuscito.

### Violazioni della sicurezza famose in TENEX

Il sistema operativo TENEX era molto diffuso sui DEC-10. Non è più usato, ma sarà ricordato per sempre negli annali della sicurezza dei computer per i seguenti errori di progettazione. TENEX utilizzava la paginazione: per permettere agli utenti di controllare il comportamento dei loro programmi era possibile fare in modo che il sistema chiamasse una funzione utente per ogni page fault.

Inoltre TENEX utilizzava le password per proteggere i file, e per accedere a un file, un programma doveva fornire al sistema la password appropriata. Il sistema operativo controllava le password, carattere per carattere, fermandosi non appena rilevava un errore. Per entrare abusivamente in TENEX, un intruso posizionava accuratamente una password, come mostrato in Figura 9.12(a), con il primo carattere alla fine di una pagina e il resto all'inizio della pagina successiva.

Il passo successivo consisteva nell'assicurarsi che la seconda pagina non fosse in memoria, per esempio facendo riferimento a così tante altre pagine che la seconda pagina sarebbe stata sicuramente scritta su disco per fare spazio. A questo punto il programma provava ad aprire il file della vittima, usando la password allineata con cura: se il primo carattere della vera password non era *A*, il sistema avrebbe interrotto il controllo dopo il primo carattere e avrebbe restituito il messaggio ILLEGAL PASSWORD. Al contrario, se la vera password cominciava con *A*, il sistema avrebbe continuato la lettura, generando un fault di pagina di cui l'intruso sarebbe stato informato.

Se la password non cominciava con *A*, l'intruso cambiava la password come quella in Figura 9.12(b) e ripeteva tutto il processo per vedere se essa cominciava con *B*; erano necessarie al più 128 prove per considerare tutto l'insieme di caratteri ASCII e determinare così il primo carattere.

Si supponga che il primo carattere fosse *F*: la configurazione della memoria di Figura 9.12(c) permetteva all'intruso di provare stringhe della forma *FA*, *FB* e così via. L'uso di

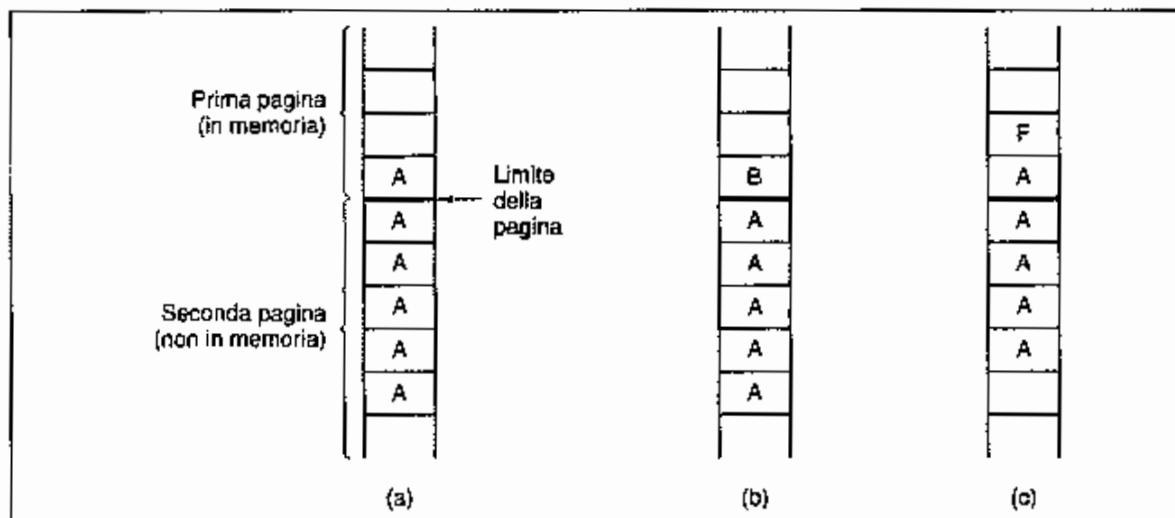


Figura 9.12 Il problema della password in TENEX.

questo approccio necessitava al più  $128^n$  prove per indovinare gli  $n$  caratteri ASCII della password invece di  $128^n$ .

## Violazioni della sicurezza famose in OS/360

Consideriamo infine i difetti dell'OS/360; la descrizione che segue è lievemente semplificata, ma preserva l'essenza del difetto. In questo sistema era possibile iniziare la lettura di un nastro e poi continuare il calcolo mentre l'unità nastro stava trasferendo i dati nello spazio utente. In questo caso, il trucco consisteva nell'iniziare la lettura del nastro, quindi fare una chiamata di sistema che richiedeva una struttura dati utente, per esempio un file da leggere e la sua password.

Il sistema operativo prima verificava che la password fosse davvero quella corretta, poi ritornava indietro e leggeva di nuovo il nome del file per l'accesso vero e proprio (avrebbe potuto salvare il nome internamente, ma non lo faceva). Sfortunatamente, poco prima che il sistema andasse a prendere il nome del file una seconda volta, esso era riscritto dal nastro; il sistema quindi leggeva il nuovo file, per il quale non era stata data nessuna password. Fare tutto ciò richiedeva un po' di pratica, ma non era poi così difficile.

## 9.4.8 Principi di progettazione per la sicurezza

Adesso dovrebbe essere chiaro che la progettazione di un sistema operativo sicuro non è una questione così semplice. Molte persone stanno lavorando su questo problema da decenni senza molto successo; sin dal 1975, i ricercatori hanno identificato alcuni principi generali che dovrebbero essere utilizzati come guida alla progettazione di sistemi sicuri (Saltzer e Schroeder, 1975). Nel seguito viene data una breve descrizione delle loro idee (basate sull'esperienza con MULTICS); queste idee risultano oggi valide, così come lo erano all'epoca.

Primo, la progettazione del sistema dovrebbe essere pubblica: il supporre che gli intrusi non sappiano come funzioni il sistema, serve solo ad illudere i progettisti! Gli intrusi prima o poi lo scopriranno, e se la protezione è compromessa da questa conoscenza, il sistema cede.

Secondo, non vi dovrebbe essere alcun accesso di default: errori in cui è rifiutato un accesso legale causeranno lamentele molto più velocemente di quegli errori in cui è consentito un accesso illegale. Nel dubbio, meglio dire "No"!

Terzo, controllo dell'autorità corrente: il sistema non dovrebbe controllare il permesso, determinare se esso sia consentito e poi mettere da parte questa informazione per un uso successivo. Molti sistemi verificano il permesso quando un file viene aperto, e non dopo: ciò significa che un utente che apre un file e lo tiene aperto per una settimana, continua ad avere accesso, anche se il proprietario ha cambiato la protezione del file o magari ha anche cercato di cancellarlo.

Quarto, dare ad ogni processo il minor privilegio possibile: se un editor ha il solo diritto di accedere ai file da editare (specificato quando il l'editor viene chiamato), editor che contengano cavalli di Troia non potranno fare molti danni. Questo principio implica uno schema di protezione a grana fine, e dà luogo a schemi che verranno discussi nel seguito del capitolo.

Quinto, il meccanismo di protezione dovrebbe essere semplice, uniforme, e fare parte della fondamenta del sistema: cercare di ripristinare la sicurezza in un sistema non sicuro già esistente è praticamente impossibile. La sicurezza, così come la correttezza, non è una caratteristica aggiuntiva!

Sesto, lo schema scelto deve essere psicologicamente accettabile: se gli utenti percepiscono come un lavoro troppo oneroso il proteggere i loro file, non lo faranno, e nondimeno, si lamentineranno molto se qualcosa va storto. Risposte del tipo "È colpa tua", generalmente non saranno ben accette.

A questa lista aggiungeremmo un altro principio ricavato da anni di dura esperienza:

#### *Mantenere il progetto semplice*

Se il sistema è elegante e semplice, è stato progettato da un singolo architetto ed ha poche linee guida che determinano il resto, vi è la possibilità che sia sicuro. Se il progetto è confuso, senza coerenza, e con molte concessioni fondamentali a vecchi sistemi non sicuri, in nome di una compatibilità verso il passato, è destinato a diventare un incubo per la sicurezza. Si può progettare un sistema con molte caratteristiche (opzioni, interfaccia utente amichevole, eccetera), ma un sistema con molte funzionalità è un grande sistema, ed un grande sistema è potenzialmente un sistema insicuro. Più codice c'è, più vi saranno buchi nella sicurezza e bachi, perciò da un punto di vista della sicurezza, il progetto più semplice è il progetto migliore.

## 9.5 Attacchi dall'esterno del sistema

Le minacce discusse nelle sezioni precedenti in gran parte erano causate dall'interno, cioè, perpetrata da utenti che avevano già ottenuto un accesso. Tuttavia per le macchine connesse ad Internet, o ad un'altra rete, c'è una crescente minaccia esterna: un computer di una rete può essere attaccato anche da un computer distante tramite la rete. In quasi tutti i casi, tale attacco consiste in codice trasmesso, tramite la rete, alla macchina bersaglio ed ivi eseguito, arrecando danni; visto che sempre più computer sono connessi ad Internet, i danni potenziali continuano ad aumentare. Nelle sezioni successive esamineremo alcuni aspetti collegati ai sistemi operativi di queste minacce esterne, focalizzandoci principalmente su virus, worm, codice mobile e applet Java.

In questi giorni è difficile aprire un giornale senza leggere dell'ennesimo virus per computer o worm che ha messo in subbuglio il mondo dei computer: rappresentano chiaramente un grande problema per la sicurezza sia degli individui sia delle aziende. Nelle sezioni successive esamineremo come agiscono e ciò che si può fare con essi.

Ho avuto qualche esitazione nello scrivere questa parte con così tanti dettagli, per paura che dia ad alcune persone delle pessime idee, ma altri libri in circolazione forniscono anche maggiori dettagli ed includono anche vero codice (ad esempio Ludwig, 1998). Anche Internet è pieno di informazioni sui virus, perciò il genio è già uscito dalla lampada! Inoltre è difficile per la gente difendersi dai virus, se non sanno come agiscono; infine vi sono molte idee sbagliate in giro e bisogna in qualche modo correggerle.

A differenza, ad esempio dei programmati di giochi, i creatori di virus di successo tendono a non cercare pubblicità, dopo il lancio del loro prodotto. Basandosi sulla scarse statistiche esistenti, pare che la maggior parte siano degli studenti di liceo o universitari o laureati da poco tempo, che hanno creato virus come una sfida tecnologica, non capendo (o non considerando) che un attacco di virus può causare tante vittime quante un uragano o un terremoto. Chiamiamo il nostro antieroe Virgil, il creatore del virus. Tipicamente, l'obiettivo di Virgil è che il virus si diffonda rapidamente, che sia difficilmente rilevabile, e che sia difficile eliminarlo, una volta che sia stato rilevato.

Cos'è dunque un virus? Per rendere breve una lunga storia, un virus è un programma

che si può riprodurre, attaccando il suo codice ad un altro programma, in maniera analoga a come si riproducono i virus biologici; inoltre può fare altre cose oltre a riprodursi. I worm sono come i virus, ma sono autoreplicanti. Questa differenza qui non ci riguarda, perciò utilizzeremo il termine "virus" per parlare di entrambi, almeno per adesso. Esamineremo i worm nella Sezione 9.5.5.

### 9.5.1 Scenari di danni causati da virus

Visto che un virus è programma, può fare tutto ciò che fa un programma. Ad esempio, può stampare un messaggio, fare comparire un'immagine sullo schermo, suonare musica, o qualsiasi altra cosa inoffensiva. Sfortunatamente può anche modificare, cancellare, distruggere, o rubare file (trasmettendoli altrove via e-mail). Il ricatto è un'altra possibilità: immaginate un virus che cripti tutti i file contenuti nel disco fisso della vittima, e poi mostri il seguente messaggio:

SALUTI DAL GENERALE CRITTAZIONE!

PER OTTENERE UNA CHIAVE DI DECRITTAZIONE PER IL TUO HARD  
DISK, MANDA 100 DOLLARI IN BANCONOTE NON SEGNATE E DI PICCOLO  
TAGLIO ALLA CASELLA POSTALE 2154, PANAMA CITY, PANAMA. GRAZIE.  
APPREZZIAMO IL TUO LAVORO.

Un'altra azione che il virus può compiere è rendere il computer inutilizzabile fino a quando il virus agisce: tale azione è detta **attacco tramite negazione del servizio**. L'approccio usuale consiste nel consumare ampiamente le risorse, come la CPU, oppure riempiendo il disco fisso di robaccia. Qui di seguito viene mostrato un programma di una linea che solitamente annienta i sistemi UNIX:

```
main() {while(1) fork();}
```

Questo programma genera processi fino a che la tabella dei processi non è piena, impedendo ad ogni altro processo di iniziare; ora immaginate un virus che infetti ogni programma del sistema con questo codice. Per prevenire questo tipo di problemi, molti sistemi UNIX moderni limitano il numero di figli che un processo può generare.

Ancora peggio, il virus può danneggiare in maniera permanente l'hardware del computer: molti computer moderni memorizzano il BIOS nella flash ROM, che può essere riscritta tramite programmi particolari (per permettere alle case produttrici di distribuire elettronicamente la correzione dei bachi); un virus può scrivere qualcosa di casuale nella flash ROM in modo che il computer non possa più effettuare l'avvio. Se il chip della flash ROM si trova in uno zoccolo, per risolvere il problema bisogna aprire il computer e sostituire il chip; se la flash ROM è saldata alla scheda, probabilmente si dovrà sostituire l'intera scheda con una nuova: in ogni caso un'esperienza poco divertente!

Un virus può anche essere creato con un obiettivo specifico: un'azienda può immettere sul mercato un virus che controlli se sia in esecuzione presso una ditta concorrente e che nessun amministratore di sistema sia collegato al momento; se il campo è libero, il virus interferirà col processo di produzione, riducendo la qualità del prodotto, causando in questo modo dei problemi alla ditta concorrente. Negli altri casi non fa nulla, ma risulterà difficile rilevarlo.

Un altro esempio di virus con obiettivo potrebbe essere quello creato dall'ambizioso vicepresidente di un'azienda e immesso nella rete locale: il virus controlla di essere atti-

vo sulla macchina del presidente e, se così fosse, cerca un foglio elettronico, ed inverte due celle casuali. Prima o poi il presidente prenderà una decisione sbagliata perché basata sui risultati del foglio elettronico, e forse sarà anche licenziato, lasciando il suo posto libero per chi sapete!

## 9.5.2 Come agiscono i virus

Visti abbastanza scenari dei danni potenziali, possiamo ora esaminare come funziona un virus. Virgil scrive un virus, probabilmente in assembler, quindi, facendo attenzione, lo inserisce in un programma nel suo computer, tramite uno strumento chiamato **dropper**; tale programma infetto viene poi diffuso, magari mandandolo ad un bulletin board o ad una raccolta di software gratis su Internet. Il programma potrebbe essere un nuovo, eccitante gioco, la versione pirata di un qualche software commerciale, o qualsiasi altro materiale che si può plausibilmente desiderare, di modo che la gente possa poi iniziare a scaricare il programma infetto.

Una volta installato sulla macchina della vittima, il virus rimane in uno stato di quiete fino a che il programma infetto non viene eseguito; quando poi viene eseguito, di solito inizia con l'infettare altri programmi presenti sulla macchina, eseguendo poi il suo vero incarico (**payload**). In molti casi tale incarico non viene eseguito fino ad una certa data, per essere sicuri che il virus si sia diffuso prima che la gente lo noti, e tale data potrebbe anche contenere un messaggio politico (ad esempio se cade nel 100mo o 500mo anniversario di una qualche grave offesa al gruppo etnico dell'autore).

Nella disamina che segue, esamineremo sette diversi tipi di virus, a seconda di ciò che viene infettato: esistono virus compagni, per programmi eseguibili, della memoria, del settore d'avvio, dei driver dei dispositivi, delle macro, e del codice sorgente. Non c'è dubbio che nuove categorie appariranno in futuro.

### Virus compagni

Un **virus compagno** non infetta realmente un programma, ma viene eseguito quando si suppone che il programma venga eseguito. Tale concetto risulta più facile a spiegarsi con un esempio. In MS-DOS, quando un utente digita

```
prog
```

MS-DOS prima cerca un programma con nome *prog.com*; se non lo trova, cerca un programma con nome *prog.exe*; in Windows, quando un utente esegue un click su Start e poi Run, accade la stessa cosa. Oggigiorno la maggior parte dei programmi sono file con estensione *.exe*, mentre quelli con estensione *.com* sono molto rari.

Supponiamo che Virgil sappia che molte persone eseguono *prog.exe* da un prompt di MS-DOS o dal Run di Windows; costui può semplicemente mettere in circolazione un virus chiamato *prog.com*, che verrà eseguito ogni volta che qualcuno cercherà di eseguire *prog* (a meno che non venga digitato il nome completo: *prog.exe*). Quando *prog.com* ha terminato il suo compito, esegue *prog.exe* e l'utente non si accorge di nulla.

Un attacco in qualche modo collegato a questo utilizza il Desktop di Windows, che contiene link simbolici dei programmi: un virus può modificare uno di tali link in modo che punti al virus stesso. Quando l'utente esegue un doppio click su un'icona, il virus viene eseguito, e quando questo ha eseguito il suo compito, lancia il programma effettivamente collegato al link.

```

#include <sys/types.h> /* * intestazione POSIX standard */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;

/* per la chiamata lstat, che verifica
 se il file è un link simbolico */

cerca(char * nomedir)
{
 DIR * dirp;
 struct dirent * dp;

 dirp = opendir(nomedir);
 if (dirp == NULL) return;
 while (TRUE) {
 dp = readdir(dirp);
 if (dp == NULL) {
 chdir("..");
 break;
 }
 if (dp->d_name[0] == '.') continue;
 lstat(dp->d_name, &sbuf);
 if (S_ISLNK(sbuf.st_mode)) continue;
 if (chdir(dp->d_name) == 0) {
 search(".");
 } else {
 if (access(dp->d_name, X_OK) == 0)
 infetta(dp->d_name);
 }
 closedir(dirp);
 }
}

```

Figura 9.13 Una procedura ricorsiva che cerca file eseguibili su sistemi UNIX.

### Virus di programmi eseguibili

Un passo avanti nella complessità è svolto dai virus che infettano i programmi eseguibili. Il più semplice di questi virus si sovrascrive al programma eseguibile; questo genere di virus sono detti **virus sovrascriventi**. La logica di tale infezione viene mostrata nella Figura 9.13.

Il programma principale di questo virus copia prima il suo programma binario su un array, aprendo *argv[0]* e leggendolo per metterlo al sicuro; quindi visita tutto il file system a partire dalla directory root, portandosi in tale directory e chiamando *cerca* con la directory root come parametro.

La procedura ricorsiva *cerca* esamina una directory aprendola, leggendo quindi le entry una alla volta tramite *readdir*, fino a che non venga restituito un NULL, che indica che non vi sono altre entry. Se l'entry è una directory, questa viene esaminata, spostandosi in essa e poi chiamando *cerca* ricorsivamente; se è un file eseguibile, viene infettato chiamando *infetta* con il nome del file da infettare come parametro. I file che cominciano con “.” vengono saltati per evitare problemi con le directory . e ..; anche i link simbolici vengono saltati perché il programma suppone di poter accedere ad una directory tramite la chiamata di sistema *chdir* e poi tornare indietro dove era andando in .., cosa che funziona con gli hard link, ma non con i link simbolici. Un programma più furbo può anche gestire i link simbolici.

La vera procedura che infetta, *infetta* (non mostrata), deve semplicemente aprire il file indicato come suo parametro, sovrascrivere sul file il virus salvato nell'array e poi chiudere il file.

Questo virus può essere "migliorato" in diversi modi: primo, si potrebbe inserire un test in *infetta* per generare un numero casuale e ritornare nella maggior parte dei casi senza fare nulla. In, diciamo, un caso su 128, l'infezione si fa davvero, riducendo così la possibilità di essere rilevata troppo presto, prima che il virus abbia avuto modo di diffondersi a sufficienza. I virus biologici hanno la stessa caratteristica: quelli che uccidono la propria vittima velocemente non si diffondono nei pressi così rapidamente come quelli che danno una morte lenta e progressiva, che però dà alla vittima la possibilità di diffondere il virus. Un progetto alternativo potrebbe essere l'avere un più alto tasso di infezione (diciamo il 25%), ma con meno file infettati ogni volta, in modo da ridurre l'attività del disco ed essere meno pesante.

Secondo, *infetta* potrebbe controllare se il file è già stato infettato: infettare due volte lo stesso file è una perdita di tempo. Terzo, si potrebbe memorizzare la data dell'ultima modifica e la dimensione del file, in modo che l'infezione rimanga nascosta: per i programmi più grandi dei virus, la dimensione rimarrà invariata, ma per i programmi più piccoli dei virus, la dimensione sarà ora maggiore. E poiché la maggior parte dei virus è più piccola della maggior parte dei programmi, questo non è un problema serio.

Sebbene questo programma non sia così lungo (il programma completo è lungo meno di una pagina in C e una volta compilato il suo testo occupa meno di 2 KB), una versione in assembler può essere anche più breve. Ludwig (1998) fornisce la versione in assembler di un programma per MS-DOS che infetta tutti i file della sua directory ed è lungo solamente 44 byte!

Più avanti nel capitolo studieremo i programmi antivirus, cioè programmi che scovano e cancellano i virus. Nondimeno è interessante notare che la logica di Figura 9.13, che un virus può utilizzare per cercare i file eseguibili da infettare, può essere anche utilizzata da un programma antivirus per scovare tutti i programmi infetti, e dunque rimuovere il virus. Le tecnologie di infezione e disinfezione vanno a braccetto, ed è questo il motivo per cui è necessario capire a fondo come funzionino i virus, perché solo così si possono combattere efficacemente.

Dal punto di vista di Virgil, il problema dei virus che sovrascrivono è che si possono rilevare troppo facilmente: dopo tutto, quando si esegue un programma infetto, può diffondere il virus ancora un po', ma non fa quello che ci si aspetta che faccia, e l'utente se ne accorgerà immediatamente. Di conseguenza la maggior parte dei virus si attacca al

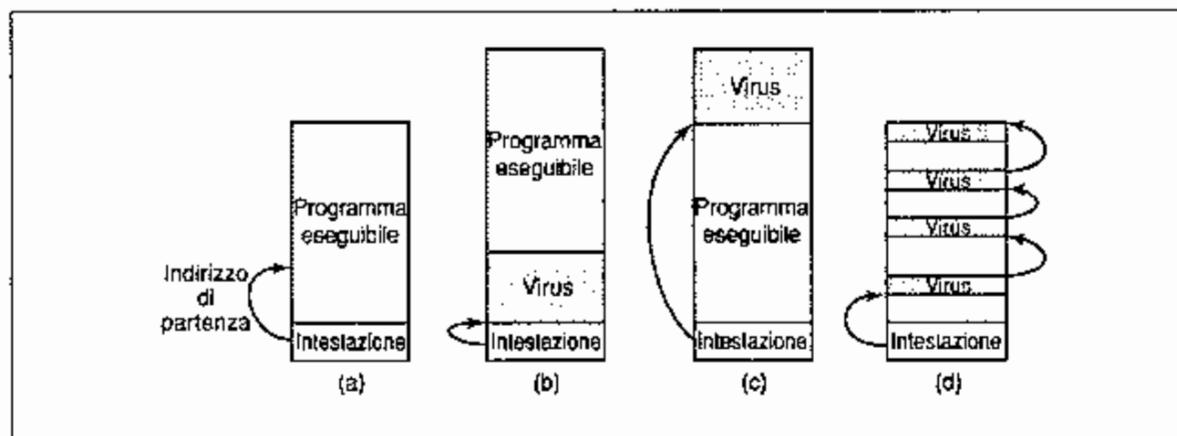


Figura 9.14 [a] Un programma eseguibile. [b] Con un virus in testa. [c] Con un virus in coda. [d] Con un virus diffuso nello spazio libero all'interno del programma.

programma e fa il suo sporco lavoro, ma permette ugualmente al programma di funzionare normalmente: tale genere di virus è chiamato **virus parassita**.

I virus parassiti possono attaccarsi in testa, in coda o nel mezzo dei programmi eseguibili: se un virus si attacca in testa, deve prima copiare il programma nella RAM, poi scrivere se stesso in cima al file ed infine ricopiare il programma dalla RAM, dietro se stesso, come mostrato in Figura 9.14(b). Sfortunatamente il programma non potrà essere eseguito dal suo nuovo indirizzo virtuale, quindi il virus deve rilocare il programma in base a dove si è spostato, oppure deve spostarlo all'indirizzo virtuale 0, dopo avere eseguito se stesso.

Per evitare le due operazioni complicate richieste a questi virus che si attaccano in testa, la maggior parte si attacca in coda, collocandosi al termine del programma eseguibile invece che in testa, inodificando il campo dell'indirizzo di partenza nell'intestazione in modo che punti all'inizio del virus, come mostrato in Figura 9.14(c). Il virus ora verrà eseguito da un indirizzo virtuale differente, a seconda del programma infetto che è in esecuzione, ma tutto ciò significa che Virgil deve essere sicuro che il suo virus sia indipendente dalla posizione, utilizzando indirizzi relativi invece di quelli assoluti, e questo non è difficile per un programmatore esperto.

I programmi eseguibili, come i file .exe in Windows e quasi tutti i formati binari moderni di UNIX, permettono ad un programma di avere un formato complesso, con molte aree testo e segmenti di dati, che il loader mette assieme in memoria e riloca all'istante. In alcuni sistemi (ad es., Windows), i segmenti (sezioni) sono multipli di 512 byte; se un segmento non è pieno, il linker lo riempie di 0. Un virus che capisca ciò può cercare di nascondersi nei buchi: se vi si inserisce completamente, come in Figura 9.14(d), la dimensione del file rimane immutata rispetto al file non infetto, chiaramente un bel vantaggio, perché un virus nascosto è un virus felice! I virus che utilizzano questo principio sono detti **virus delle cavità**. Ovviamente se il loader non carica le zone vuote in memoria, il virus avrà bisogno di un altro mezzo per essere eseguito.

## Virus residenti in memoria

In precedenza abbiamo supposto che quando viene eseguito un programma infetto, il virus agisce, passa il controllo al vero programma e poi termina. A differenza di ciò, un **virus residente in memoria** rimane in memoria tutto il tempo, o nascondendosi nella parte alta della memoria, o nella parte bassa in mezzo ai vettori di interruzione, le poche ultime centinaia di byte solitamente non utilizzati. Un virus molto intelligente può anche modificare le mappe di bit della RAM del sistema operativo, facendo in modo che il sistema pensi che la memoria del virus sia occupata, evitando così il pericolo di essere sovrascritto.

Un tipico virus residente in memoria cattura uno dei vettori di trap o interruzione, copiandone il contenuto in una variabile temporanea ed inserendovi il proprio indirizzo, in modo che la trap o l'interruzione punti ad esso. La scelta migliore è la chiamata di sistema trap: in questo modo il virus può essere eseguito (in modo kernel) ad ogni chiamata di sistema; quando avviene ciò, chiama la vera chiamata di sistema, saltando all'indirizzo della trap salvato.

Perché un virus vorrebbe essere eseguito ad ogni chiamata di sistema? Per infettare programmi, naturalmente! Il virus può aspettare finché viene eseguita una chiamata di sistema exec, quindi, saputo che il file a portata di mano è un file binario eseguibile (e probabilmente utile per i suoi scopi), lo infetta. Questo processo non richiede il massiccio utilizzo del disco di Figura 9.13, così risulta molto meno evidente; inoltre il fatto di poter catturare tutte le chiamate di sistema, dà al virus la capacità potenziale di poter spiare i dati e compiere tutti i generi di misfatti.

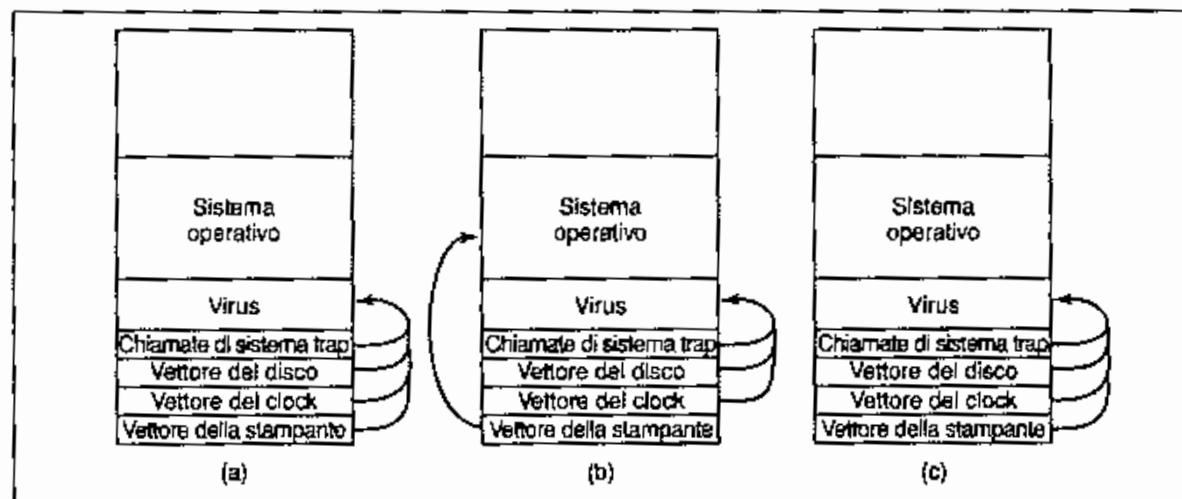
## Virus del settore d'avvio

Come abbiamo discusso nel Capitolo 5, quando la maggior parte dei computer viene accesa, il BIOS legge il record di avvio principale dall'inizio del disco di boot nella RAM e lo esegue; questo programma determina quale partizione sia attiva e legge il primo settore, il settore d'avvio, da tale partizione e lo esegue. Quindi tale programma o carica il sistema operativo o utilizza un loader per caricare il sistema operativo. Sfortunatamente molti anni fa, un amico di Virgil ebbe l'idea di creare un virus che potesse sovrascrivere il record di avvio principale o il settore d'avvio, ottenendo risultati devastanti. Tali virus, detti **virus del settore d'avvio**, sono molto comuni.

Di norma un virus del settore d'avvio [che contiene i virus dell'MBR (Master Boot Record, il record di avvio principale)] dapprima copia il vero settore d'avvio in un posto sicuro del disco, in modo da poter poi caricare il sistema operativo quando ha terminato di agire. Il programma di formattazione del disco della Microsoft, fdisk, salta la prima traccia, così essa risulta un buon posto in cui nascondersi nelle macchine dotate di Windows. Un'altra possibilità è utilizzare un qualsiasi settore vuoto e poi modificare la lista dei settori danneggiati, marcando il nascondiglio come danneggiato: infatti se il virus è grande, può anche camuffarsi tra i settori danneggiati; anche la fine della directory radice è una possibilità, se tale directory è abbastanza grande ed è situata in un posto fisso, come accade in Windows 98. Un virus davvero aggressivo potrebbe anche allocare del normale spazio su disco per il vero settore d'avvio e per se stesso, aggiornando poi in maniera conforme la mappa di bit o la lista dello spazio vuoto del disco: fare ciò richiede una profonda conoscenza delle strutture dati interne del sistema operativo, ma Virgil ha avuto un bravo professore nel corso di sistemi operativi, ed ha studiato molto.

Quando viene avviato il computer, il virus copia se stesso nella RAM, o in cima o tra i vettori di interruzione non utilizzati; a questo punto la macchina si trova in modo kernel, con la MMU spenta, nessun sistema operativo, e nessun programma antivirus in esecuzione: una vera festa per i virus! Pertanto quando è pronto, avvia il sistema operativo, che di solito si trova nella memoria residente.

Il problema adesso è come ottenere il controllo più tardi: la maniera usuale è sfruttare specifiche conoscenze di come il sistema operativo gestisca i vettori di interruzione. Ad esempio Windows non sovrascrive tutti i vettori di interruzione in un colpo solo; al con-



**Figura 9.15** (a) Dopo che il virus ha catturato tutti i vettori di interruzione e di trap. (b) Dopo che il sistema operativo ha ripreso il vettore dell'interruzione della stampante. (c) Dopo che il virus ha capito di avere perso il vettore dell'interruzione della stampante e lo ha ricatturato.

trario carica i driver dei dispositivi uno alla volta, ed ognuno cattura il vettore di interruzione di cui necessita. Questo processo può richiedere un minuto.

Questo tipo di architettura dà al virus l'aiuto di cui necessita: incomincia col catturare tutti i vettori di interruzione come mostrato in Figura 9.15(a); non appena i driver sono stati caricati, alcuni dei vettori sono sovrascritti, ma a meno che il driver del clock non sia caricato prima, vi saranno in seguito abbastanza interruzioni del clock che daranno il via al virus. La perdita dell'interruzione della stampante è mostrata in Figura 9.15(b): non appena il virus si accorge che uno dei suoi vettori di interruzione è stato sovrascritto, può sovrascrivere ancora tale vettore, perché adesso è sicuro (in realtà alcuni vettori sono sovrascritti diverse volte durante l'avvio, ma il meccanismo è deterministico e Virgil lo sa a memoria). La ricattura dell'interruzione della stampante è mostrata in Figura 9.15(c). Quando tutto è stato caricato, il virus ripristina tutti i vettori di interruzione e tiene per sé solo il vettore della chiamata di sistema trap. Dopo tutto ottenere il controllo ad ogni chiamata di sistema è molto più divertente che ottenere il controllo dopo ogni operazione del floppy disk, ma durante l'avvio, non può correre il rischio di perdere il controllo per sempre. A questo punto abbiamo un virus residente in memoria che controlla le chiamate di sistema, infatti è così che prende vita la maggior parte dei virus residenti in memoria.

### Virus del driver dei dispositivi

Penetrare nella memoria in questo modo assomiglia un poco alla speleologia - bisogna procedere contorcendosi e preoccupandosi di ciò che può piovere dall'alto, finendo sulla testa: sarebbe inoltre più semplice se il sistema operativo caricasse gentilmente il virus in maniera legale. Con uno piccolo sforzo aggiuntivo, si può raggiungere tale scopo: il trucco consiste nell'infettare un driver di dispositivo, portando ad un virus di driver di dispositivo. In Windows ed alcuni sistemi UNIX, i driver di dispositivo sono programmi eseguibili che risiedono su disco e sono caricati al momento dell'avvio; se uno di questi può essere infettato con un virus parassita, il virus verrà sempre caricato in maniera legale all'avvio: anche meglio, i driver agiscono in modo kernel e dopo che un driver è stato caricato, viene chiamato il virus, dando ad esso la possibilità di catturare il vettore della chiamata di sistema trap.

### Virus delle macro

Molti programmi, ad esempio Word e Excel, permettono all'utente di scrivere macro per raggruppare diversi comandi, che si potranno poi eseguire con un singolo tasto. Si possono associare macro anche ad elementi di menu, perciò quando ne viene scelto uno, la macro viene eseguita. In Microsoft Office, le macro possono contenere interi programmi scritti in Visual Basic, un linguaggio di programmazione completo; le macro vengono interpretate piuttosto che compilate, ma questo influenza solo sulla velocità di esecuzione e non su ciò che possono fare. Visto che le macro possono essere specifiche di un documento, Office memorizza le macro di ogni documento assieme al documento stesso.

Il problema nasce a questo punto: Virgil scrive un documento in Word e crea una macro che associa alla funzione OPEN FILE, e tale macro contiene un virus della macro. Egli spedisce poi via e-mail il documento alla vittima, che naturalmente lo apre (supponendo che il programma di e-mail non lo abbia già fatto per lui); l'azione di aprire il file fa sì che la macro OPEN FILE venga eseguita, e siccome la macro può contenere un programma arbitrario, può fare qualunque cosa, come infettare altri documenti Word, can-

cellare file e così via. Per essere giusti nei confronti di Microsoft, *Word* avvisa quando si sta aprendo un file con macro, ma molti utenti non sanno di cosa si tratti e procedono nell'aprire comunque il file. Inoltre anche documenti legali possono contenere macro, e vi sono anche programmi che non forniscono tale avviso, rendendo anche più difficile il rilevare un virus.

Col crescere dell'uso di e-mail, spedire documenti con virus inglobati in macro diventa un problema immane: tali virus si possono scrivere più facilmente che nascondendo il vero settore d'avvio da qualche parte nella lista dei settori danneggiati, o celando il virus tra i vettori di interruzione e catturando il vettore della chiamata di sistema trap. Ciò significa che persone sempre meno preparate possono creare dei virus, abbassando la qualità generale del prodotto e dando una cattiva nomea ai creatori di virus.

## Virus del codice sorgente

I virus parassiti e del settore d'avvio sono strettamente dipendenti dalla piattaforma; i virus dei documenti lo sono un po' meno (*Word* funziona su Windows e Macintosh, ma non su UNIX); i virus maggiormente portabili sono i virus del codice sorgente. Immaginate il virus di Figura 9.13, ma con una modifica: invece di ricercare file binari eseguibili, va alla ricerca di programmi C; questo richiede una modifica di una riga soltanto (la chiamata `ad access`). La procedura `infetta` dovrebbe essere modificata inserendo

```
#include <virus.h>
```

all'inizio di ogni programma sorgente C. È necessario inserire un'altra riga,

```
run_virus();
```

per attivare il virus. Decidere dove inserire questa linea richiede una certa abilità nell'analisi del codice C, perché deve essere un posto che permetta sintatticamente di effettuare chiamate alla procedura, e non un posto in cui il codice non servirebbe (ad esempio dopo un `return`). Anche inserire tale linea nel mezzo di un commento non funziona, mentre metterla all'interno di un ciclo potrebbe essere troppo. Supponendo che la chiamata sia nella posizione appropriata (ad esempio, proprio prima del termine del `main` o prima del `return` se ve ne è uno), una volta che il programma è stato compilato, contiene il virus, preso da `virus.h` (anche se `proj.h` potrebbe attrarre meno l'attenzione se qualcuno dovesse notarlo).

Quando il programma è in esecuzione, chiama il virus: esso può fare qualunque cosa desideri, ad esempio, cercare altri programmi C da infettare; e se ne trova uno, può includere le due linee date precedentemente, ma questo funzionerà solo sulla macchina locale, dove si suppone che `virus.h` sia già stato installato. Per fare sì che funzioni anche su una macchina remota, bisogna includere tutto il codice sorgente del virus: si può realizzare ciò includendo il codice sorgente del virus come una stringa di caratteri inizializzata, preferibilmente come una lista di interi esadecimali a 32 bit per evitare che qualcuno possa capire cosa faccia. Questa stringa sarà probabilmente abbastanza lunga, ma col codice di milioni di righe che va di moda oggi, potrebbe facilmente passare inosservata.

Ad un lettore alle prime armi, tutte queste descrizioni possono apparire abbastanza complicate e ci si potrebbe legittimamente chiedere se si riesce a farle funzionare in pratica. Ebbene, si riesce: Virgil è un eccellente programmatore ed ha molto tempo libero. Controlla sul tuo giornale locale e ne avrai la prova!

### 9.5.3 Come si diffondono i virus

Esistono diversi scenari di diffusione; cominciamo con quello classico: Virgil crea il suo virus, lo inserisce in alcuni programmi che ha scritto (o rubato), ed inizia a distribuire il programma, ad esempio, mettendolo in siti Web specializzati in shareware; alla fine qualcuno scarica il programma e lo esegue. Vi sono diverse opzioni a questo punto: tanto per cominciare, il virus probabilmente infetta più file sul disco fisso, nel caso in cui la vittima decida di condividere alcuni di questi con un amico in seguito. Può anche cercare di infettare il settore d'avvio del disco fisso; una volta infettato il settore d'avvio, è facile avviare un virus residente in memoria in modo kernel nei successivi avvii.

Inoltre il virus può controllare se vi siano dei floppy disk nei drive, e in caso vi siano, infettare i loro file ed i loro settori d'avvio: i floppy disk rappresentano un buon obiettivo, perché sono trasferiti da una macchina all'altra più frequentemente dei dischi fissi. Se il settore d'avvio di un floppy disk è infetto e tale dischetto viene utilizzato per avviare un'altra macchina, può iniziare ad infettare i file ed il settore d'avvio del disco fisso di tale macchina. Nel passato, quando erano la via principale di trasmissione dei programmi, questo meccanismo era il modo in cui generalmente si diffondevano i virus.

Oggi giorno Virgil ha a disposizione altre opzioni: il virus può controllare se la macchina infetta sia connessa ad una LAN, cosa che è molto probabile per una macchina che appartenga ad un'azienda o ad una università. Il virus può quindi iniziare ad infettare i file non protetti dei server connessi alla LAN; questa infezione non toccherà i file protetti, ma può riuscirci facendo comportare in maniera strana i programmi infetti: un utente che esegua tali programmi chiederà probabilmente aiuto all'amministratore di sistema; l'amministratore proverà anch'egli il programma, per vedere cosa stia accadendo. Se l'amministratore fa ciò mentre ha i diritti di superuser, il virus può infettare i codici binari del sistema, i driver dei dispositivi, il sistema operativo ed i settori d'avvio. Basta un solo errore come questo e tutte le macchine della LAN sono compromesse.

Spesso le macchine di una LAN hanno l'autorizzazione di accedere a macchine remote in Internet o in aziende private, o anche l'autorizzazione ad eseguire comandi in remoto senza eseguire il login: questa possibilità fornisce al virus più opportunità di diffondersi; in questo modo un errore innocente può infettare l'intera azienda. Per evitare questo scenario, tutte le aziende dovrebbero adottare una politica generale che suggerisca agli amministratori di non commettere mai errori.

Un altro modo per diffondere un virus è spedire un programma infetto ad un newsgroup di USENET o ad un bulletin board al quale si mandano regolarmente programmi. È anche possibile creare una pagina Web che richiede una speciale plug-in per essere visualizzata dal browser, e distribuire delle plug-in infette.

Un diverso attacco consiste nell'infettare un documento e poi mandarlo via e-mail a molte persone o ad una mailing list o ad un newsgroup di USENET, di solito come allegato: anche le persone che non si sognerebbero mai di eseguire un programma che uno sconosciuto ha mandato loro, potrebbero non sapere che eseguire un click per aprire un allegato può diffondere un virus sulla loro macchina. A peggiorare la situazione, il virus può cercare la rubrica dell'utente e spedire e-mail a chiunque compaia nella rubrica, di solito con un Subject che sembra realistico o interessante, come

```
Subject: Cambio' di programmi
Subject: Re: la tua ultima mail
Subject: Il cane è morto la scorsa notte
Subject: Sono seriamente malato
Subject: I love you
```

Quando l'e-mail arriva, il destinatario vede che il mittente è un amico o un collega, non sospetta che vi siano problemi ed una volta che la mail è stata aperta, è troppo tardi. Il virus "I LOVE YOU" che si è diffuso nel mondo, nel giugno del 2000, agiva proprio in questo modo, ed ha causato miliardi di dollari di danni.

La diffusione della tecnologia dei virus è in qualche modo collegata alla diffusione effettiva di virus attivi: vi sono gruppi di creatori di virus che comunicano attivamente in Internet e si aiutano vicendevolmente per sviluppare nuove tecnologie, strumenti, e virus. La maggior parte di questi probabilmente lo fa più per hobby che come lavoro criminale, ma gli effetti possono essere ugualmente devastanti. Un'altra categoria di creatori di virus sono i militari, che considerano i virus come un'arma da guerra, potenzialmente in grado di rendere inefficienti i computer del nemico.

Un altro problema legato alla diffusione dei virus è come evitare che siano rilevati: le prigioni di norma non hanno buona attrezzature informatiche, perciò Virgil preferirebbe evitarle. Se spedisce il virus iniziale dalla suo computer di casa, corre un certo rischio; se l'attacco ha successo, la polizia potrebbe risalire a lui cercando tra i messaggi del virus quello col timestamp più vecchio, poiché probabilmente è quello più vicino alla fonte dell'attacco.

Per diminuire la sua esposizione, Virgil potrebbe andare ad un Internet Cafe in una città distante dalla sua e effettuare lì un login: può portare con sé un floppy disk e leggerlo, oppure, se non tutte le macchine hanno il drive per il floppy disk, può chiedere alla graziosa signorina al banco se può leggere il file *book.doc*, in modo che lo possa stampare. Una volta che il file è sul disco fisso, lo rinomina *virus.exe* e lo esegue, infettando l'intera LAN con un virus che entra in azione due settimane più tardi, nel caso in cui la polizia decida di chiedere alle compagnie aeree la lista di tutte le persone che hanno volato in quella settimana. Un metodo alternativo consiste nel tralasciare il floppy disk e prendere il virus da un sito FTP, oppure portare un portatile e collegarlo ad una porta Ethernet o USB che l'Internet Cafe ha premurosamente fornito per quei turisti muniti di portatile, che vogliono leggere la loro e-mail ogni giorno.

### 9.5.4 Tecniche antivirus ed anti-antivirus

I virus cercano di nascondersi, mentre gli utenti cercano di scovarli, e tutto ciò porta ad un gioco tipo gatto col topo; diamo quindi una occhiata ad alcune di tali questioni. Per evitare di essere visibili quando si listano le directory, un virus compagno, un virus del codice sorgente, o un altro file che non dovrebbe trovarsi lì, possono portare a uno il bit HIDDEN in Windows o utilizzare un nome di file che inizi con "." in UNIX. In modo più sofisticato si può modificare *explorer* di Windows o *ls* di UNIX per evitare che siano listati quei file il cui nome inizia con *Virgil-*. I virus si possono nascondere anche in posti inconsueti o insospettabili, come la lista dei settori danneggiati su disco o il registro di Windows (un database in memoria disponibile per programmi, per memorizzare stringhe non interpretate). La flash ROM utilizzata per memorizzare il BIOS e la memoria CMOS sono altre possibilità, sebbene sia difficile scrivere nella prima e la seconda sia molto piccola; e naturalmente la principale fucina del mondo dei virus è infettare i programmi eseguibili ed i documenti presenti sull'hard disk.

#### Analizzatori di virus

Chiaramente l'utente medio non riuscirà a scovare molti virus che fanno del loro meglio per nascondersi, per questo motivo si è sviluppato un mercato di software antivi-

rus, e qui sotto spiegheremo come funziona questo software. Le aziende che producono software antivirus posseggono laboratori in cui scienziati specializzati passano molte ore per rilevare e capire i nuovi virus: il primo passo consiste nel fare sì che un virus infetti un programma che non fa niente, spesso chiamato *goat file* (file capra), per ottenere una copia del virus nella sua forma più pura. Il passo successivo consiste nel generare un esatto elenco del codice del virus ed inserirlo nel database dei virus conosciuti: le aziende concorrono tra loro sulla dimensione dei loro database, ed inventare nuovi virus per gonfiare il proprio database, non è considerato sportivo.

Una volta che programma antivirus è stato installato sulla macchina di un cliente, la prima cosa che fa è analizzare ogni file eseguibile del disco, andando alla ricerca di ogni virus presente nel database dei virus conosciuti. Molte aziende che producono antivirus hanno un sito Web, da cui i clienti possono scaricare le descrizioni dei virus scoperti di recente dai loro database: se l'utente ha 10000 file e il database contiene 10000 virus, naturalmente è necessaria una programmazione intelligente per rendere più veloce la ricerca.

Siccome nel tempo compaiono delle varianti dei virus noti, è necessaria una ricerca fuzzy, di modo che una modifica di 3 byte ad un virus, non gli permetta di sfuggire alla rilevazione. Tuttavia le ricerche fuzzy non solo sono più lente delle ricerche precise, ma possono anche generare falsi allarmi, cioè, avvisi di pericolo per file innocui che casualmente contengono codice vagamente simile a quello di un virus segnalato in Pakistan sette anni prima. Cosa si suppone che l'utente faccia trovandosi di fronte al messaggio:

WARNING! Il file xyz.exe può contenere il virus Lahore-9x. Cancellare?

Più sono i virus nel database e più generale è il criterio per dichiarare un successo, maggiori saranno i falsi allarmi: se sono troppi, l'utente si arrenderà disgustato; ma se l'analizzatore dei virus insiste su una corrispondenza quasi perfetta, può lasciarsi sfuggire alcuni virus modificati. Raggiungere la giusta misura rappresenta un (instabile) equilibrio euristico: teoricamente, il laboratorio dovrebbe identificare alcune parti essenziali del virus che probabilmente non cambiano, ed utilizzarle per riconoscere il virus da analizzare, come se fossero le loro "firme".

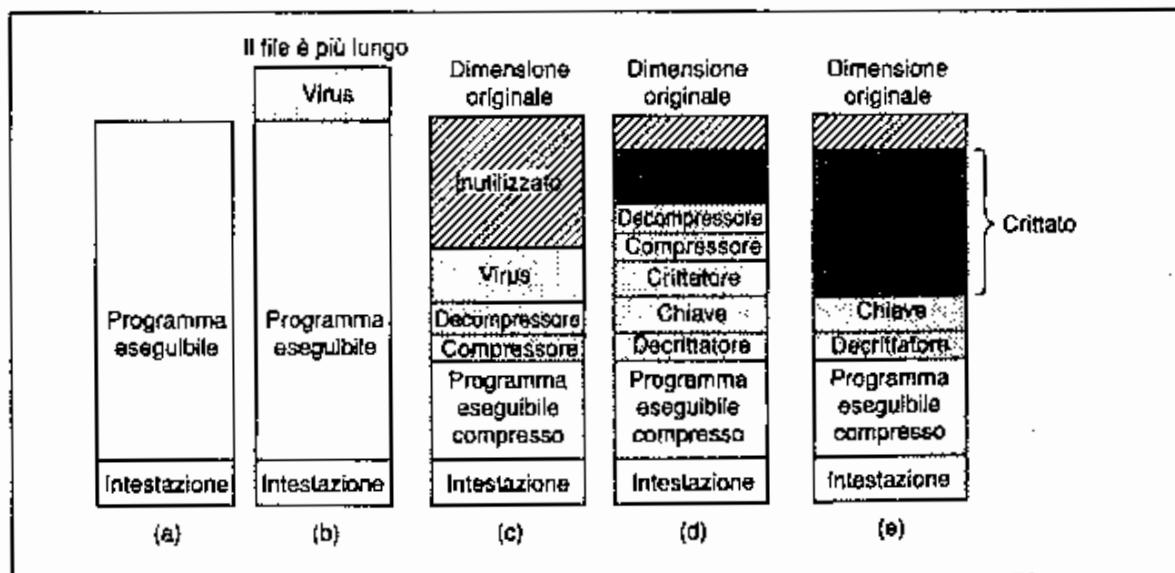


Figura 9.16 [a] Un programma. [b] Un programma infetto. [c] Un programma compresso infetto. [d] Un virus crittato [e] Un virus compresso con il codice di compressione crittato.

Se il disco è stato dichiarato privo di virus la scorsa settimana, non significa che lo sia ancora; perciò l'analizzatore di virus deve essere utilizzato frequentemente, ma, visto che l'analisi richiede tempo, è più efficiente controllare solo quei file che sono stati modificati dalla data dell'ultima analisi. Il problema è che un virus intelligente azzererà la data di un file infetto, riportandola al suo valore originale, per evitare che sia rilevato: la risposta del programma antivirus è controllare la data dell'ultima modifica della directory che contiene tale file. E la risposta del virus è modificare la data della directory e questo è l'inizio del gioco del gatto col topo a cui alludevamo prima.

Un altro modo in cui il programma antivirus rileva l'infezione di file è registrare e memorizzare su disco la lunghezza di tutti i file: se un file è cresciuto dall'ultimo controllo, potrebbe essere stato infettato, come mostrato in Figura 9.16(a-b); tuttavia un virus intelligente può evitare di essere rilevato comprimendo il programma e riportando il file alla sua lunghezza originale. Per fare sì che questo schema funzioni, il virus deve contenere le procedure sia di compressione che di decompressione, come mostrato in Figura 9.16(c).

Un altro modo in cui il virus cerca di fuggire alla rilevazione è quello di essere sicuro che la sua rappresentazione su disco non corrisponda per niente a quella presente nel database del software antivirus. Un modo per raggiungere tale obiettivo è crittare se stesso con una chiave diversa per ogni file infetto: prima di fare una nuova copia, il virus genera una chiave di crittazione a 32 bit casuale, ad esempio eseguendo lo XOR del tempo corrente con il contenuto, diciamo, delle parole 72008 e 319992 della memoria. Quindi esegue lo XOR del suo codice con questa chiave, parola per parola, per produrre il virus crittato, memorizzato nel file infetto, come mostrato in Figura 9.16(d); la chiave è memorizzata nel file. Per propositi di segretezza, mettere la chiave nel file non è l'ideale, ma qui l'obiettivo è confondere l'analizzatore di virus e non evitare che gli appositi scienziati del laboratorio antivirus risalgano al codice sorgente del virus. Naturalmente, per agire, il virus deve prima decrittare se stesso; perciò avrà bisogno anche di una procedura di decrittazione nel file.

Questo schema non è perfetto perché le procedure di compressione, decompressione, crittazione e decrittazione sono le stesse in tutte le copie; perciò il programma antivirus può utilizzarle per riconoscere il virus da analizzare. Nascondere le procedure di compressione, decompressione, crittazione e decrittazione è semplice: esse sono crittate con il resto del virus, come mostrato in Figura 9.16(e); tuttavia il codice di decrittazione non può essere crittato, ma deve essere veramente eseguito sull'hardware per decrittare il resto del virus, e dunque deve essere presente in chiaro. I programmi antivirus lo sanno, e perciò danno la caccia alla procedura di decrittazione.

Tuttavia Virgil vuole avere l'ultima parola, e procede come segue. Supponiamo che la procedura di decrittazione debba eseguire il calcolo

$$X = (A + B + C - 4)$$

Il semplice codice assembler per questo calcolo, per un computer generico con due indirizzi, è mostrato in Figura 9.17(a). Il primo indirizzo è la sorgente; il secondo è la destinazione, perciò MOV A,R1 sposta la variabile A nel registro R1. Il codice in Figura 9.17(b) fa la stessa cosa, solo in maniera meno efficiente a causa dell'istruzione NOP (no operation), inserita nel vero codice.

Ma non siamo ancora a posto; è possibile caniuffare il codice di decrittazione. Esistono molti modi per rappresentare NOP, ad esempio, aggiungendo 0 ad un registro, eseguendo l'OR con se stesso, shiftando a sinistra di 0 posizioni, e saltando all'istruzione successiva: tutto ciò non fa nulla. Così il programma di Figura 9.17(c) è funzionalmente

|           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
| MOV A,R1  |
| ADD B,R1  | NOP       | ADD #0,R1 | OR R1,R1  | TST R,1   |
| ADD C,R1  | ADD B,R1  | ADD B,R1  | ADD B,R1  | ADD C,R1  |
| SUB #4,R1 | NOP       | OR R1,R1  | MOV R1,R5 | MOV R1,R5 |
| MOV R1,X  | ADD C,R1  | ADD C,R1  | ADD C,R1  | ADD B,R1  |
|           | NOP       | SHL #0,R1 | SHL R1,0  | CMP R2,R5 |
|           | SUB #4,R1 | SUB #4,R1 | SUB #4,R1 | SUB #4,R1 |
|           | NOP       | JMP .+1   | ADD R5,R5 | JMP .+1   |
|           | MOV R1,X  | MOV R1,X  | MOV R1,X  | MOV R1,X  |
|           |           |           | MOV R5,Y  | MOV R5,Y  |
| (a)       | (b)       | (c)       | (d)       | (e)       |

Figura 9.17 Esempio di virus polimorfo.

lo stesso di Figura 9.17(a); quando copia se stesso il virus potrebbe usare la Figura 9.17(c) invece della Figura 9.17(a) ed in seguito agire ancora quando eseguito. Un virus che muta ad ogni copia è chiamato **virus polimorfo**.

Ora supponiamo che R5 non sia necessario in questo pezzo di codice; allora anche Figura 9.17(d) è equivalente alla Figura 9.17(a); infine, in molti casi è possibile invertire le istruzioni senza modificare ciò che il programma fa, perciò concludiamo con la Figura 9.17(e), che è un frammento di codice logicamente equivalente alla Figura 9.17(a). Un pezzo di codice che può cambiare la sequenza delle istruzioni macchina senza modificare la sua funzionalità è detto **motore di mutazione**, e i virus sofisticati li contengono per modificare il derivatore da una copia all'altra. Lo stesso motore di mutazione può essere nascosto, crittandolo assieme al corpo del virus.

Chiedere al povero software antivirus di capire che le Figure dalla 9.17(a) alla 9.17(e) sono tutte funzionalmente equivalenti è chiedere molto, specialmente se il motore di mutazione ha molti assi nelle sue maniche. Il software antivirus può analizzare il codice per vedere cosa faccia, e può anche cercare di simularne le operazioni, ma bisogna tenere presente che può avere migliaia di virus e migliaia di file da analizzare, e dunque non ha molto tempo per ogni test a meno di essere terribilmente lento.

Tra l'altro, l'istruzione che memorizza nella variabile Y è stata inserita proprio per rendere più difficile rilevare il fatto che il codice collegato a R5 sia codice inutile, cioè, che non fa nulla: se altri frammenti di codice leggono o scrivono Y, il codice rimarrà perfettamente legale. Un motore di mutazione ben scritto può generare un buon codice polimorfo che può causare gli incubi a chi scrive software antivirus, ma il lato positivo è che un tale motore è difficile da scrivere; perciò gli amici di Virgil usano tutti il suo codice, il che significa che non ce ne sono molti differenti in circolazione- non ancora, almeno.

In precedenza abbiamo parlato di come cercare di riconoscere i virus nei file eseguibili infetti; inoltre l'analizzatore antivirus deve controllare l'MBR, il settore di avvio, la lista dei settori danneggiati, la flash ROM, la memoria CMOS, eccetera, ma cosa succede se un virus residente in memoria è attualmente in azione? Non sarà rilevato. O ancora peggio, supponiamo che il virus in azione stia monitorando le chiamate di sistema: può facilmente rilevare che il programma antivirus sta leggendo il settore d'avvio (per cercare virus); quindi per contrastare il programma antivirus, il virus non esegue la chiamata di sistema, ma restituisce invece il vero settore d'avvio, recuperandolo dal suo nascondiglio nella lista dei settori danneggiati. Si fa anche un appunto per ri-infettare tutti i file quando l'analizzatore di virus ha terminato.

Per evitare di essere notato da un virus, il programma antivirus potrebbe fare letture fisiche su disco, aggirando il sistema operativo; tuttavia ciò richiede di avere già pronti i driver dei dispositivi per dischi IDE, SCSI e gli altri tipi più comuni, rendendo il programma antivirus meno portabile e soggetto a fallimenti sui quei computer dotati di dischi non usuali. Inoltre, visto che è possibile aggirare il sistema operativo per leggere il settore d'avvio, ma aggirarlo per leggere tutti i file eseguibili non lo è, c'è anche il rischio che il virus possa creare dati falsi a proposito dei file eseguibili.

## I controlli di integrità

Un approccio completamente differente nel rilevare i virus è il **controllo di integrità** (integrity checking). Un programma antivirus che agisce in questo modo, prima analizza il disco fisso alla ricerca di virus e, una volta che è convinto che il disco è pulito, calcola un checksum per ogni file eseguibile e scrive la lista dei checksum per tutti i file rilevanti nella directory in un file, *checksum*, anch'esso nella directory. La prossima volta che viene eseguito, ricalcola tutti i checksum e controlla se essi corrispondono a quanto compare nel file *checksum*: un file infetto sarà immediatamente visibile.

Il problema è che Virgil non ha intenzione di fare accadere ciò; egli può creare un virus che cancella il file dei checksum, o ancora peggio, può creare un virus che calcoli il checksum dei file infetti e sostituisca i vecchi valori nel file di checksum. Per proteggersi da questo genere di comportamento, il programma antivirus può cercare di nascondere il file di checksum, ma probabilmente non funzionerà, perché Virgil può studiare attentamente il comportamento del programma antivirus prima di creare il virus. Un'idea migliore è crittarlo per fare sì che le manomissioni siano più facili da rilevare; teoricamente, la crittazione dovrebbe coinvolgere l'utilizzo di una smart card con una chiave memorizzata esternamente che i programmi non possono raggiungere.

## I controlli di comportamento

Una terza strategia utilizzata dai software antivirus è il **controllo del comportamento** (behavioral checking). Con questo approccio, il programma antivirus risiede in memoria mentre il computer è in funzione, e cattura tutte le chiamate di sistema: l'idea è che possa monitorare tutte le attività e cerchi quindi di catturare tutto ciò che appare sospetto. Ad esempio, nessun programma normale dovrebbe cercare di sovrascrivere il settore d'avvio, e dunque un tentativo di fare ciò è quasi certamente dovuto a un virus; similmente, anche modificare la flash ROM è fortemente sospetto.

Ma vi sono casi che sono meno evidenti: ad esempio, sovrascrivere un file eseguibile è un'azione strana da fare - a meno di non essere un compilatore. Se il software antivirus rileva una tale scrittura e segnala il pericolo, l'utente (sperabilmente) sa se sovrascrivere un eseguibile ha senso o meno nel contesto del lavoro corrente; allo stesso modo, se Word sovrascrive un file .doc con un nuovo documento pieno di macro, non è necessariamente opera di un virus. In Windows, i programmi possono staccarsi dal loro file eseguibile e diventare residenti in memoria tramite una chiamata di sistema speciale; ancora una volta questo potrebbe essere legale, ma una segnalazione può essere utile.

I virus non devono andare in giro passivamente nell'attesa di un programma antivirus che li uccida, come bestie destinate al macello, ma possono combattere. Può scoppiare una battaglia particolarmente interessante se un virus residente in memoria, si imbatte in un antivirus residente in memoria sul medesimo computer: anni fa esisteva un gioco, chiamato Core Wars, in cui due programmati si affrontavano mettendo entrambi un

programma in uno spazio di indirizzi vuoto; i programmi esploravano a turno la memoria, e lo scopo del gioco era localizzare e annientare il proprio nemico, prima che fosse questo ad annientarvi. Il confronto virus-antivirus assomiglia un po' a questo gioco, solo che il campo di battaglia è il computer di un povero utente che non vuole davvero che vi accada ciò. Ancora peggio, il virus ha un vantaggio, perché il suo creatore può scoprire molto circa il programma antivirus, semplicemente comprandone una copia; ma, naturalmente, una volta che il virus è in circolazione, la squadra antivirus può modificare il suo programma, obbligando Virgil a comprarsene una nuova copia.

## Evitare i virus

Ogni storia che si rispetti necessita di una morale; la morale di questa è

*Meglio sicuro che dispiaciuto.*

Evitare i virus è in primo luogo molto più semplice che snidarli una volta che abbiano infettato un computer. Qui sotto diamo alcune indicazioni per gli utenti, ma anche alcune cose che le industrie possono fare per ridurre considerevolmente il problema.

Che cosa possono fare gli utenti per evitare l'infezione di un virus? Primo, scegliere un sistema operativo che dia un elevato grado di sicurezza, con un preciso confine tra i modi kernel ed utente, e password e login separate per ogni utente ed amministratore di sistema. In queste condizioni, un virus che in qualche modo entra di nascosto non può infettare i codici binari del sistema.

Secondo, installare solo software confezionato ed acquistato da una ditta affidabile; anche questa non è una garanzia, in quanto vi sono stati dei casi in cui impiegati di cattivo umore hanno immesso dei virus in prodotti software commerciali, ma comunque aiuta molto. Scaricare software da siti Web e bulletin board è un comportamento rischioso.

Terzo, comprare un buon pacchetto software antivirus ed utilizzarlo secondo le istruzioni; controllando di ricevere gli aggiornamenti periodici dal sito Web della ditta produttrice.

Quarto, non fare click sugli allegati delle e-mail, e dire in giro di non spedirteli. Le e-mail spedite in formato ASCII testuale sono sempre sicure, mentre gli attachment possono avviare un virus al momento in cui vengono aperti.

Quinto, fare frequenti backup dei file chiave su un supporto esterno, come floppy disk, CD riscrivibile, o nastro. Tenere diverse generazioni per ogni file su una serie di backup, in questo modo se si scopre un virus, si può avere la possibilità di recuperare i file nella forma in cui erano prima di essere infettati. Recuperare i file di ieri che sono stati infettati non è d'aiuto, mentre recuperare le versioni della settimana precedente potrebbe servire.

Anche l'industria dovrebbe prendere seriamente la minaccia del virus e modificare alcune abitudini pericolose. Primo, rendere i sistemi operativi semplici: più campanelli e fischietti vi sono, più buchi nella sicurezza sono presenti, è un fatto della vita.

Secondo, lasciar perdere i documenti attivi, che, da un punto di vista della sicurezza sono un disastro. Visualizzare un documento che qualcuno ha spedito non dovrebbe richiedere di eseguire tale programma: i file con estensione JPEG, ad esempio, non contengono programmi e dunque non possono contenere virus. Tutti i documenti dovrebbero funzionare in questo modo.

Terzo, dovrebbe esistere un modo per proteggere selettivamente da scrittura cilindri specificati del disco, in modo da evitare che i virus infettino i programmi presenti su que-

sti. Questa protezione potrebbe essere realizzata avendo una mappa di bit all'interno del controllore che lista i cilindri protetti da scrittura; la mappa dovrebbe essere modificabile solo quando l'utente fa scattare un interruttore meccanico presente sul pannello frontale del computer.

Quarto, la flash ROM è una buona idea, ma dovrebbe essere modificabile solo quando sia scattato un interruttore esterno, cosa che accadrà solamente quando l'utente sta consapevolmente installando un aggiornamento del BIOS. Naturalmente nessuno di questi punti verrà preso seriamente in considerazione, finché non avrà colpito un potente virus, ad esempio, un virus che colpisca il mondo finanziario e azzeri tutti i conti bancari: naturalmente, a quel punto sarebbe troppo tardi.

### Riprendersi da un attacco di un virus

Quando un virus viene rilevato, il computer dovrebbe essere bloccato immediatamente in quanto potrebbe esserci ancora in azione un virus residente in memoria; il computer dovrebbero essere riavviato da un CD-ROM o un floppy disk, sempre protetto in scrittura, e che contengono il sistema operativo completo, in modo da aggirare i settori d'avvio, la copia su disco fisso del sistema operativo ed i driver del disco, tutti i quali potrebbero essere infetti. Quindi si dovrebbe eseguire un programma antivirus, partendo dal suo CD-ROM originale, in quanto anche la copia presente su disco fisso potrebbe essere infetta.

Il programma antivirus può rilevare alcuni virus ed anche essere in grado di eliminarli, ma non vi sono garanzie che li troverà tutti. Probabilmente la sequenza di azioni più sicura da fare a questo punto è salvare tutti i file che non possono contenere virus (come i file ASCII e JPEG), mentre quei file che potrebbero contenere virus (come file Word), dovrebbero essere convertiti in un altro formato che non possa contenere virus, come il formato testo ASCII (o almeno dovrebbero essere eliminate le macro); infine tutti questi file dovrebbero essere salvati su un supporto esterno. Quindi si dovrebbe riformattare il disco fisso, utilizzando un programma di formattazione preso da un floppy disk protetto in scrittura o un CD-ROM, per assicurarsi che non siano essi stessi infetti; ed è particolarmente importante che l'MBR e i settori d'avvio vengano cancellati; infine si dovrebbe reinstallare il sistema operativo con il CD-ROM originale. Quando si ha a che fare con le infezioni di virus, la paranoia è il miglior amico.

### 9.5.5 Il Worm di Internet

La prima grande violazione su vasta scala via Internet, della sicurezza dei computer cominciò la sera del 2 novembre 1988, quando un dottorando di Cornell, Robert Tappan Morris, lanciò un programma **worm** (verme) nella rete Internet. Questo fatto provocò il crollo di migliaia di computer nell'università, nelle aziende e nei laboratori governativi in tutto il mondo prima di essere scoperto e rimosso. Inoltre tutto ciò diede vita a una discussione non ancora terminata; vedremo in seguito i momenti culminanti di questo evento. Per informazioni più tecniche si guardi Spafford, 1989; e per la storia vista come un giallo poliziesco si guardi Hafner e Markoff, 1991.

La storia cominciò nel 1988, quando Morris scoprì due errori nello UNIX di Berkeley che davano la possibilità di ottenere l'accesso non autorizzato alle macchine di tutta la rete Internet. Lavorando da solo, egli scrisse un programma che si autoriproduceva chiamato **worm**, che avrebbe sfruttato questi errori e si sarebbe riprodotto in pochi secondi su ogni macchina di cui avesse potuto ottenere l'accesso. Egli lavorò per mesi sul pro-

gramma, mettendolo a punto con cura per fare sì che non lasciasse tracce della sua esistenza.

Non si sa se la versione del 2 novembre 1988 doveva essere un test o il vero attacco; ad ogni modo essa mise in ginocchio moltissimi sistemi SUN e VAX di Internet nel giro di poche ore dalla sua esecuzione. Non si conoscono le ragioni di Morris, ma è possibile che per lui l'intera faccenda fosse solo uno scherzo di alta tecnologia, ma che a causa di un errore di programmazione gli sfuggì completamente dalle mani.

Tecnicamente il worm era composto da due programmi, il bootstrap e il worm vero e proprio: il bootstrap era di 99 linee di C chiamato *l1.c*; era compilato ed eseguito sul sistema sottoposto ad attacco. Una volta lanciato, si collegava alla macchina da cui proveniva, caricava il worm principale e lo eseguiva. Dopo aver risolto qualche problema per nascondere la sua esistenza, il worm ispezionava le tabelle di routing della sua nuova macchina per vedere a quali macchine era collegata, e tentava di trasmettere il bootstrap a queste macchine.

Per infettare nuove macchine venivano provati tre metodi: il primo consisteva nel provare a lanciare una shell remota usando il comando *rsh*; alcune macchine si fidano delle altre macchine e spontaneamente eseguono *rsh* senza ulteriori verifiche di autenticità. Se ciò funzionava, la shell remota caricava il programma worm e continuava a infettare le nuove macchine da lì.

Il secondo metodo usava un programma presente su tutti i sistemi BSD chiamato *finger* che permette a qualsiasi utente della rete Internet di lanciare il comando

```
finger name@site
```

per visualizzare informazioni su una persona in una particolare installazione. Questa informazione di solito contiene il vero nome della persona, la login, il numero di telefono e l'indirizzo di casa e di lavoro, il nome e il numero di telefono della segretaria, il numero di fax e altre informazioni simili. È l'equivalente elettronico di una rubrica telefonica.

Il comando *finger* funziona come segue: ad ogni sito BSD, un processo di background detto *finger daemon* (demone di finger) è sempre attivo per ricevere e rispondere a domande provenienti da tutta la rete Internet; quello che faceva il worm era chiamare *finger* con una stringa speciale composta da 536 byte. Questa stringa era così lunga da eccedere la dimensione del buffer del demone, e ne invadeva lo stack, nel modo mostrato in Figura 9.11(c). In questo caso si sfruttava l'errore del demone che non controllava l'overflow: quando il demone ritornava dalla procedura in cui era al momento della richiesta, non ritornava al *main*, ma a una procedura nella stringa di 536 byte sullo stack. Questa procedura provava a eseguire *sh*, ed in caso di successo il worm aveva ora una shell in esecuzione sulla macchina sottoposta ad attacco.

Il terzo metodo dipendeva da un errore nel sistema di posta, *sendmail*, che permetteva al worm di spedire una copia del bootstrap ed eseguirlo.

Una volta insediato, il worm provava a scoprire la password degli utenti; per mettere a punto ciò, Morris non dovette lavorare molto: tutto quello che dovette fare, fu chiedere a suo padre, un esperto di sicurezza presso la National Security Agency, l'agenzia governativa di sicurezza del codice degli Stati Uniti, la ristampa di un articolo classico sull'argomento che Morris Sr. e Ken Thompson avevano scritto dieci anni prima ai Bell Labs (Morris e Thompson, 1979). Ogni password individuata permetteva al worm di connettersi con tutte le macchine su cui il proprietario della password aveva un account.

Ogni volta che il worm otteneva l'accesso a una nuova macchina, controllava se vi fossero altre copie del worm già attive; in questo caso la nuova copia usciva, tranne che una

volta su sette in cui continuava l'esecuzione, forse nel tentativo di lasciare che il worm si propagasse nel caso in cui il sistemiista avesse lanciato una versione locale del worm per confondere il worm reale. L'uso di 1 tentativo su 7 creava troppi worm ed era la ragione per cui tutte le macchine infette si fermavano: erano infestate dai worm. Se Morris non avesse lasciato questa possibilità e avesse fatto uscire il worm ogni volta che intercettava un altro worm, probabilmente il worm non sarebbe stato scoperto.

Morris fu scoperto quando uno dei suoi amici parlò con John Markoff, un giornalista esperto in computer del *New York Times*, e provò a convincere Markoff che l'incidente era casuale, il worm era inoffensivo e l'autore era dispiaciuto. L'amico, inavvertitamente, si lasciò sfuggire che il login di chi aveva provocato il danno era *rtm*: passare da *rtm* al nome del proprietario fu facile; tutto quello che si doveva fare era lanciare *finger*. Il giorno dopo la storia era in un articolo in prima pagina e metteva in ombra persino le elezioni presidenziali che si sarebbero tenute tre giorni dopo.

Morris fu giudicato e condannato dalla corte federale al pagamento di una multa di 10000 dollari, a 3 anni con la sospensione condizionale della pena e a 400 ore di lavoro in una comunità. I costi legali superarono probabilmente 150000 dollari. Questa sentenza suscitò una notevole quantità di controversie: molte persone nel mondo dei computer affermavano che Morris era uno studente brillante che aveva perso il controllo di uno scherzo inoffensivo, niente nel worm lasciava pensare che Morris volesse rubare o fare danni; altri pensavano che fosse un pericoloso criminale e che meritasse di finire in galera.

Un effetto permanente di questo incidente fu la costituzione del CERT (Computer Emergency Response Team), che rappresenta la sede principale a cui riferire i tentativi di intrusione, ed un gruppo di esperti che analizza i problemi di sicurezza e progetta le soluzioni. Se questa azione rappresenta certamente un passo in avanti, ha anche un punto debole: il CERT raccoglie informazioni sui difetti dei sistemi che possono essere attaccati e come rimediare ad essi; necessariamente queste informazioni passano tra migliaia di amministratori di sistema tramite Internet, ma sfortunatamente, i cattivi (probabilmente spacciandosi per amministratori di sistema) possono essere anche in grado di ottenere informazioni sui bachi e sfruttare le scappatoie per quelle ore (o anche giorni), che trascorrono prima che siano chiuse.

## 9.5.6 Codice mobile

Virus e worm sono programmi che penetrano in un computer senza che il proprietario ne sia a conoscenza e contro la volontà dello stesso; tuttavia, qualche volta la gente più o meno consapevolmente, inserisce ed esegue sulle proprie macchine del codice sconosciuto. Di solito va in questo modo. Nel passato (che, nel mondo di Internet, significa l'anno scorso) la maggior parte delle pagine Web erano file HTML statici, a cui venivano associate poche immagini. Oggigiorno, sempre più pagine Web contengono piccoli programmi chiamati applet: quando viene scaricata una pagina Web contenente applet, le applet vengono caricate ed eseguite. Ad esempio, un'applet potrebbe contenere un modulo da compilare, con un aiuto interattivo per la compilazione; quando il modulo è stato completato, può essere mandato tramite Internet in qualche luogo dove sarà analizzato. Moduli per il pagamento delle tasse, per ordinare prodotti e molti altri tipi di moduli traggono beneficio da questo approccio.

Un altro esempio in cui i programmi vengono trasferiti da una macchina ad un'altra per essere eseguiti sulla macchina di destinazione sono gli agenti: questi sono programmi lanciati dall'utente per svolgere qualche compito, e poi restituire dei risultati. Ad esempio, si potrebbe chiedere ad un agente di cercare alcuni siti Web specializzati in viag-

gi e trovare la tratta Roma-S.Francisco più economica; arrivato ad ogni sito, l'agente entrerebbe in azione, trovando le informazioni di cui necessita, per poi spostarsi al sito Web successivo. Fatto ciò, potrebbe tornare indietro e riportare ciò che ha recuperato.

Un terzo esempio di codice mobile è un file PostScript che deve essere stampato su una stampante apposita: un file PostScript è in effetti un programma scritto nel linguaggio di programmazione PostScript, che viene eseguito all'interno della stampante; normalmente ordina alla stampante di disegnare alcune curve e poi riempirle, ma può fare qualsiasi altra cosa voglia. Applet, agenti e file PostScript sono tre esempi di codice mobile, ma ve ne sono molti altri.

Data la lunga discussione su virus e worm vista in precedenza, dovrebbe essere chiaro che permettere a codice sconosciuto di agire sulla vostra macchina è più che rischioso. Nondimeno, alcune persone vogliono eseguire questi programmi sconosciuti, così la domanda che ci si pone è: "Si può eseguire il codice mobile in modo sicuro"? Una risposta breve è: "Sì, ma non è semplice". Il problema è che quando un processo importa un'applet o altro codice mobile nel suo spazio di indirizzi e lo esegue, quel codice agisce come un processo legale dell'utente, ed ha tutti i diritti che ha l'utente, tra cui la possibilità di leggere, scrivere, cancellare e crittare i file sul disco dell'utente, spedire dati tramite e-mail in paesi lontani, e molto altro.

Tempo fa, i sistemi operativi svilupparono il concetto di processo, per creare delle divisioni tra utenti: l'idea è che ogni processo abbia il suo spazio di indirizzi corretto ed il suo UID, che gli consente di manipolare i file e le altre risorse che gli appartengono, ma non quelle degli altri utenti. Il concetto di processo però non è d'aiuto nel fornire una protezione contro una parte del processo (l'applet) ed il resto: i thread permettono thread multipli di controllo all'interno del processo, ma non fanno niente per proteggere un thread dall'altro.

In teoria, eseguire ogni applet come un processo a se stante aiuterebbe un poco, ma spesso non è fattibile; ad esempio, una pagina Web può contenere due o più applet che interagiscono tra di loro e con i dati presenti sulla pagina Web. Anche il browser potrebbe avere bisogno di interagire con le applet, avviandole o fermandole, fornendo loro dati e così via. Se ogni applet è inserita in un processo diverso, l'insieme non funzionerà; inoltre, mettere un'applet nel suo spazio di indirizzi non rende più complicato per l'applet rubare o danneggiare i dati; anzi, è più facile perché nessuno ci sta guardando dentro.

Sono stati proposti ed implementati diversi nuovi metodi per gestire le applet (ed il codice mobile in generale); qui sotto esamineremo tre di questi metodi: sandboxing, interpretazione e firma del codice, ognuno dei quali ha i suoi pregi ed i suoi difetti.

## Sandboxing

Il primo metodo, chiamato **sandboxing**, cerca di confinare ogni applet entro un intervallo limitato di indirizzi virtuali a run-time (Wahbe et al., 1993). Funziona dividendo lo spazio di indirizzamento virtuale in regioni di uguali dimensioni, che saranno chiamate **sandbox**; ogni sandbox deve godere della proprietà che tutti i suoi indirizzi condividono qualche stringa di bit in posizione più significativa. Avendo uno spazio di indirizzamento a 32 bit, potremmo dividerlo in 256 sandbox di 16 MB, di modo che tutti gli indirizzi di un sandbox abbiano in comune gli 8 bit più significativi; oppure, potremmo dividerlo in 512 sandbox di 8 MB, dove ogni sandbox ha un prefisso degli indirizzi di 9 bit. La dimensione del sandbox dovrebbe essere scelta larga abbastanza da poter contenere la più grande applet, senza perdere troppo spazio di indirizzamento virtuale. La memoria fisica non rappresenta un problema, se è presente la paginazione a richiesta, come di solito accade; vengono assegnate due sandbox ad ogni applet,

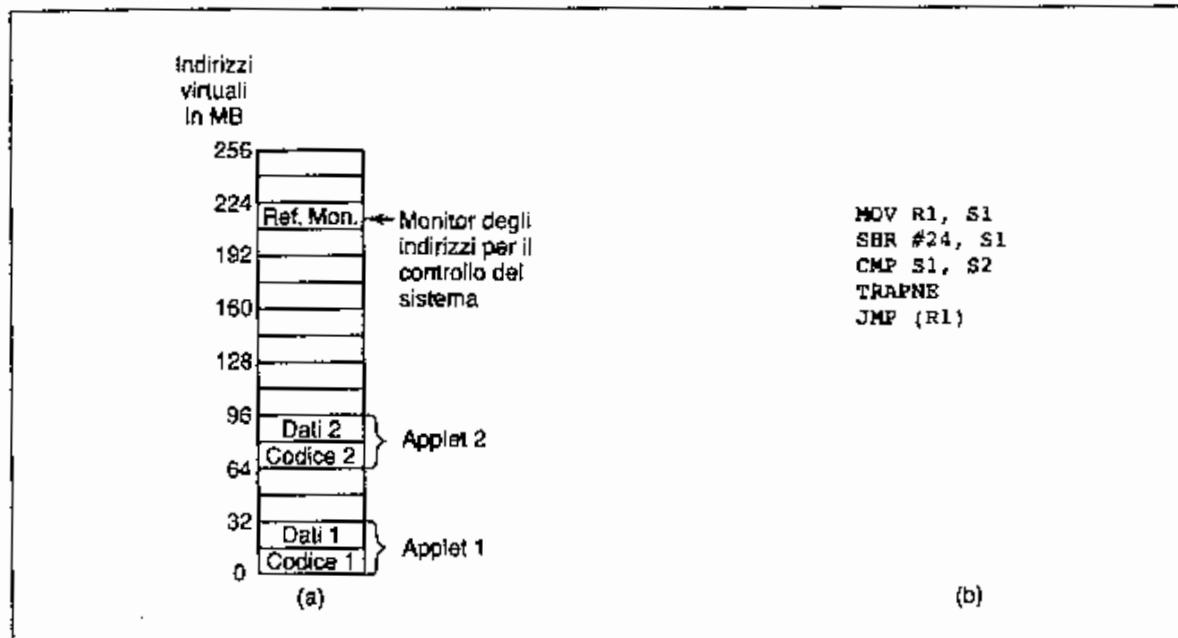


Figura 9.18 (a) Memoria divisa in sandbox da 16 Mb. (b) Un modo per controllare la validità di un'istruzione.

una per il codice, l'altra per i dati, come illustrato nella Figura 9.18(a), nel caso di 16 sandbox di 16 MB.

L'idea fondamentale che sta dietro ai sandbox è garantire che l'applet non possa saltare a codice al di fuori del sandbox per il suo codice e non possa fare riferimenti a dati al di fuori del sandbox per i suoi dati. La ragione per cui si hanno due sandbox è evitare che un'applet modifichi il suo codice durante l'esecuzione aggirando queste restrizioni. Vietando tutte le memorizzazioni nel sandbox per il codice, eliminiamo il pericolo di codice che si auto-modifica; finché l'applet è limitata in questo modo, non può causare danni né al browser, né ad altre applet, né immettere virus in memoria, o causare altri-mimenti danni alla memoria.

Non appena caricata, l'applet viene rilocata in modo che cominci all'inizio del suo sandbox; quindi vengono svolti dei controlli per vedere se i riferimenti di codice e dati sono confinati all'interno dei sandbox appositi. Nella discussione che segue, daremo un'occhiata ai riferimenti del codice (ad esempio, le istruzioni JMP e CALL), ma gli stessi discorsi valgono per i riferimenti dei dati. Le istruzioni statiche JMP che utilizzano l'indirizzamento diretto sono facili da controllare: l'indirizzo di arrivo cade all'interno dei confini del sandbox per il codice? Anche i JMP relativi sono facili da controllare: se l'applet ha un codice che cerca di lasciare il sandbox per il codice, questa viene rifiutata e non eseguita. Allo stesso modo, i tentativi di raggiungere dati al di fuori del sandbox per i dati fanno sì che l'applet venga rifiutata.

La parte difficile sono i JMP dinamici: molte macchine hanno un'istruzione in cui l'indirizzo a cui saltare è calcolato a tempo di esecuzione, messo in un registro e poi vi si salta in modo indiretto; ad esempio con JMP(R1), per saltare all'indirizzo contenuto nel registro 1. La correttezza di tali istruzioni deve essere controllata a tempo di esecuzione e viene effettuata inserendo codice direttamente prima del salto indiretto, per testare l'indirizzo di arrivo. Un esempio di un tale test è mostrato in Figura 9.18(b). Ricordiamo che tutti gli indirizzi corretti hanno gli stessi  $k$  bit più significativi, e perciò il prefisso può essere memorizzato in un registro a parte, diciamo S2. Tale registro non può essere utilizzato dalla stessa applet, il che può richiedere di riscriverla per evitare questo registro.

Il codice funziona come segue: in primo luogo, l'indirizzo di destinazione sotto esame viene copiato in un registro temporaneo, S1; quindi questo registro è shiftato a destra del giusto numero di bit per isolare il prefisso comune in S1; poi, il prefisso isolato viene confrontato al prefisso corretto, inizialmente caricato in S2; se non sono uguali, viene chiamata una trap e l'applet è uccisa. Questa sequenza di codice richiede quattro istruzioni e due registri temporanei.

Mettere delle pezze al programma binario durante l'esecuzione richiede un po' di lavoro, ma è fattibile; sarebbe più facile se l'applet si presentasse nel suo formato sorgente e poi fosse compilata localmente, utilizzando un compilatore affidabile che controlli automaticamente gli indirizzi statici ed inserisca il codice per verificare quelli dinamici durante l'esecuzione. In ogni modo, c'è un po' di lavoro in più a tempo di esecuzione associato ai controlli dinamici: Wahbe e al. (1993) hanno quantificato questo, in un 4% in più, che è generalmente accettabile.

Un secondo problema che deve essere risolto è, cosa succede quando un'applet cerca di effettuare una chiamata di sistema? La soluzione è semplice: la chiamata di sistema viene sostituita da chiamata ad un modulo speciale, chiamato **monitor degli indirizzi** (reference monitor), nella stessa fase di controlli che inserisce i controlli dell'indirizzo dinamico (oppure, se è disponibile il codice sorgente, collegando una libreria speciale che chiama il monitor degli indirizzi invece delle chiamate di sistema). Ad ogni modo, il monitor degli indirizzi esamina ogni tentativo di chiamata e decide se è sicura da eseguire: se la chiamata è giudicata accettabile, come scrivere un file temporaneo in una specifica directory temporanea, la chiamata può procedere; se si sa che la chiamata è pericolosa, oppure se il monitor degli indirizzi non lo può stabilire, l'applet viene uccisa. Infine, se un monitor degli indirizzi può stabilire quale applet lo abbia chiamato, allora basta un unico monitor degli indirizzi, presente in qualche parte della memoria, per gestire le richieste di tutte le applet; e di solito può conoscere i permessi da un file di configurazione.

## Interpretazione

Il secondo metodo per eseguire applet non sicure è eseguirle interpretandole e non lasciando loro il controllo dell'hardware: questo è l'approccio utilizzato dai Web browser. Le applet presenti sulle pagine Web di solito sono scritte in Java, che è un comune linguaggio di programmazione, oppure in linguaggio di script di alto livello, come TCL o Javascript. Le applet Java sono dapprima compilate per una macchina virtuale stack-oriented, chiamata **JVM** (Java Virtual Machine): sono proprio queste applet JVM che sono inserite nelle pagine Web; quando sono scaricate, vengono inserite in un interprete JVM all'interno del browser, come mostrato in Figura 9.19.

Il vantaggio di eseguire codice interpretato rispetto a codice compilato è che ogni istruzione viene esaminata dall'interprete prima di essere eseguita, e questo dà all'interprete la possibilità di controllare che l'indirizzo sia valido. Inoltre vengono catturate ed interpretate anche le chiamate di sistema; come siano gestite queste chiamate è un problema della politica di sicurezza. Ad esempio, se un'applet è affidabile (ad esempio proviene dal disco locale), le sue chiamate di sistema possono procedere senza problemi. Tuttavia, se un'applet non è affidabile (ad esempio proviene da Internet), può essere messa in quello che effettivamente è un sandbox, in modo da limitare le sue azioni.

Si possono interpretare anche i linguaggi di script di alto livello: qui non si usano indirizzi macchina, e quindi non si corre il pericolo che uno script cerchi di accedere alla memoria in modo illecito. Il lato negativo dell'interpretazione consiste nel fatto che generalmente è molto lenta se paragonata all'eseguire codice macchina compilato.

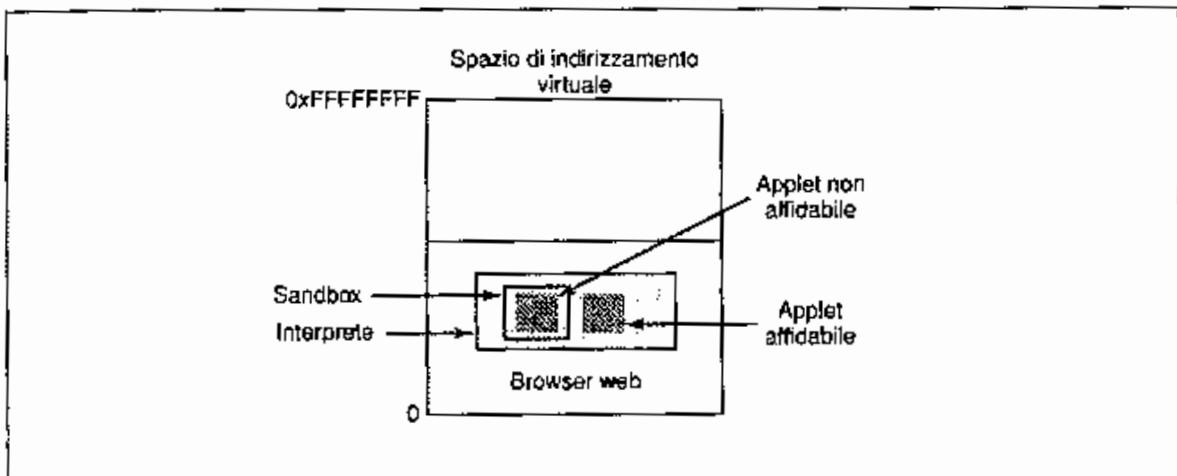


Figura 9.19 Le applet possono essere interpretate da un Web browser.

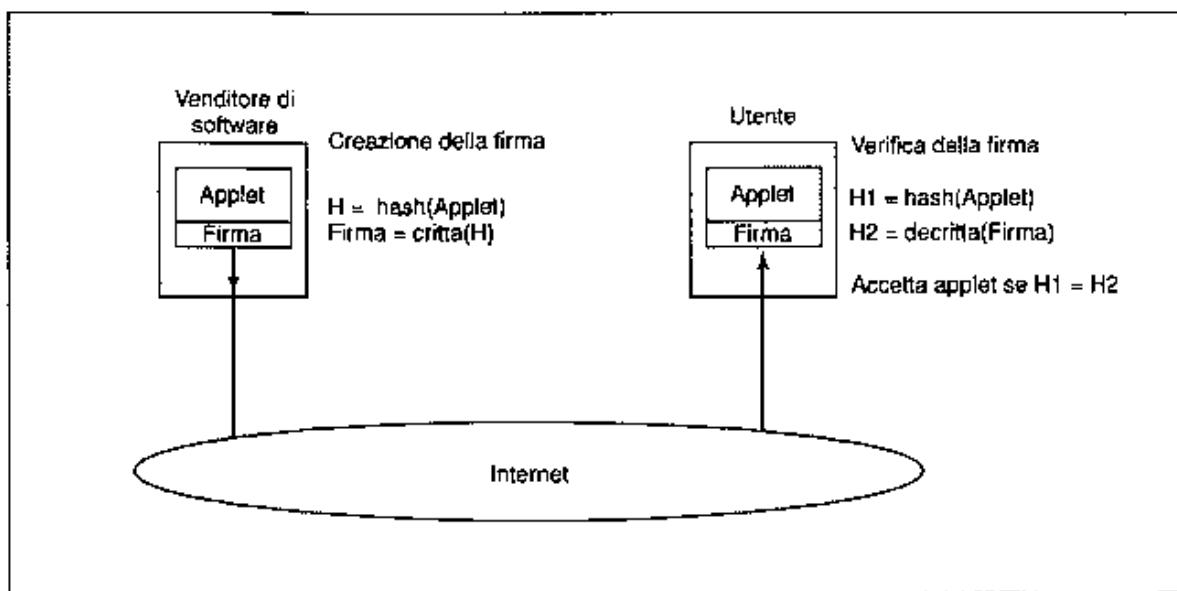
## La firma del codice

Un altro modo per gestire la sicurezza delle applet è sapere da dove provengano e accettare solo quelle provenienti da fonti sicure. Con questo approccio, un utente può mantenere una lista di fornitori di applet sicure ed eseguire solamente le applet provenienti da questi, mentre tutte le altre fonti saranno rifiutate in quanto troppo rischioso. In questo approccio non è presente alcun meccanismo di sicurezza a tempo di esecuzione: le applet provenienti da fonti sicure sono eseguite come sono, mentre il codice proveniente da altre fonti non viene eseguito per niente oppure solo in modo ristretto (tramite sandbox o interpretazione, avendo poco o nessun accesso ai file dell'utente e alle altre risorse del sistema).

Per fare funzionare questo schema, come minimo, deve esistere un modo in cui l'utente possa determinare che un'applet è stata scritta da una fonte sicura e non è stata modificata dopo la sua realizzazione: si realizza ciò tramite una firma digitale, che permette al venditore di firmare l'applet in un modo tale che le future modifiche possono essere rilevate.

La firma del codice (code signing) è basata sulla crittografia a chiave pubblica: un venditore di applet, generalmente un'azienda che produce software, genera una coppia (chiave pubblica, chiave privata), rendendo pubblica la prima e custodendo gelosamente la seconda. Per firmare un'applet, il venditore dapprima calcola una funzione hash dell'applet per ottenere un numero a 128 o 160 bit, a seconda dell'utilizzo di MD5 o SHA; poi firma il valore dell'hash crittandolo con la sua chiave privata (in realtà decrittandolo usando la notazione di Figura 9.3). Questa firma accompagnerà l'applet ovunque vada.

Quando l'utente prende l'applet, il browser calcola la funzione hash; dekritta poi la firma che la accompagna, usando la chiave pubblica del venditore e confronta quello che il venditore sostiene sia la funzione hash con quanto il browser ha calcolato: se sono uguali, l'applet è accettata come valida, altrimenti viene rifiutata come contraffatta. I calcoli matematici coinvolti rendono eccessivamente difficile per chiunque manipolare l'applet in modo che sua funzione hash corrisponda alla funzione hash ottenuta dekrittando la vera firma; è ugualmente complicato creare una nuova firma falsa che corrisponda, senza avere a disposizione la chiave privata. Il processo di firma e verifica è mostrato in Figura 9.20.



**Figura 9.20** Come funziona la firma del codice.

### 9.5.7 Sicurezza in Java

Il linguaggio di programmazione Java e il sistema a tempo di esecuzione che lo accompagna sono stati progettati per permettere ad un programma di essere scritto e compilato una volta, e di essere poi trasferito su Internet in forma binaria ed eseguito su ogni macchina che supporti Java. La sicurezza faceva parte del progetto Java sin dall'inizio, ed in questa sezione descriveremo come funzioni.

Java è un linguaggio type-safe, il che significa che il compilatore rifiuterà ogni tentativo di utilizzare una variabile in una maniera non compatibile con il suo tipo. Al contrario, consideriamo il seguente codice C:

```
funz_birichina()
{
 char *p;
 p = rand();
 *p = 0;
}
```

Qui si genera un numero casuale e lo si memorizza nel puntatore *p*; quindi si memorizza un byte a 0 all'indirizzo contenuto in *p*, sovrascrivendo qualunque cosa vi fosse, codice o dati. In Java, i costrutti che mischiano tipi in questo modo sono vietati dalla grammatica; inoltre Java non ha puntatori, cast, allocazioni di memoria controllate dall'utente (come *malloc* e *free*) e tutti i riferimenti agli array sono controllati a tempo di esecuzione.

I programmi Java sono compilati in un codice binario intermedio chiamato JVM (Java Virtual Machine) byte code; JVM ha circa 100 istruzioni, la maggior parte delle quali mettono oggetti di un tipo specifico nello stack, oppure li prelevano dallo stack, oppure combinano aritmeticamente due oggetti sullo stack. Questi programmi JVM sono tipicamente interpretati, anche se, in alcuni casi, possono essere compilati in linguaggio macchina per una esecuzione più rapida. Nel modello Java, le applet spedite su Internet per un'esecuzione remota sono programmi JVM.

Quando un'applet arriva, viene eseguita attraverso un controllore di JVM byte code che verifica se l'applet obbedisce a certe regole; un'applet compilata nella maniera appropriata obbedisce a tali regole, ma non si può fare nulla per impedire ad un utente malvagio di scrivere un'applet JVM in assembler JVM. Il controllo comprende:

1. L'applet cerca di falsificare i puntatori?
2. Viola le restrizioni negli accessi ai membri di una classe privata?
3. Cerca di utilizzare una variabile di un certo tipo come se fosse di un altro tipo?
4. Genera overflow o underflow nello stack?
5. Converte in maniera illegale una variabile da un tipo ad un altro?

Se l'applet passa tutti questi test, può essere eseguita in modo sicuro, senza temere che acceda a zone di memoria che non le spettano.

Tuttavia le applet possono effettuare delle chiamate di sistema, chiamando i metodi (procedure) Java, forniti per tale scopo; il modo in cui Java tratta tale questione si è evoluto nel tempo: nella prima versione di Java, **JDK 1.0 (Java Development Kit)**, le applet erano divise in due classi, sicure e non sicure. Le applet caricate dal disco locale erano sicure e potevano effettuare qualsiasi chiamata di sistema volessero; al contrario, le applet provenienti da Internet erano non sicure, erano eseguite in un **sandbox**, come mostrato in Figura 9.19, e non avevano il permesso di fare praticamente nulla.

Dopo qualche esperienza con questo modello, Sun decise che era troppo restrittivo. In **JDK 1.1** veniva impiegata la firma del codice: quando un'applet proveniva da Internet, veniva effettuato un controllo per verificare se era firmata da una persona o una organizzazione fidata per l'utente (il quale aveva definito la lista delle persone fidate). Se lo era, all'applet era consentito di fare qualunque cosa volesse; se non lo era, veniva eseguita in un **sandbox** e con severe restrizioni.

Dopo molte esperienze anche questo si rivelò non soddisfacente, così il modello della sicurezza fu di nuovo cambiato: **JDK 1.2** fornisce una politica di sicurezza configurabile ed a grana variabile, che si applica a tutte le applet, locali e remote. Tale modello è abbastanza complicato da poter scrivere un intero libro dedicato ad esso (Gong, 1999), perciò ne riassumeremo brevemente alcuni dei punti principali.

Ogni applet è caratterizzata da due cose: da dove viene e chi l'ha firmata. Per "da dove viene", si intende il suo URL, mentre con "chi l'ha firmata", si intende quale chiave privata è stata utilizzata per la firma. Ogni utente può creare una politica di sicurezza che consiste in una serie di regole: una regola può contenere un URL, colui che firma, un oggetto, ed una azione che l'applet può compiere sull'oggetto se l'URL dell'applet e il firmatario corrispondono alla regola. Concettualmente, le informazioni fornite sono mostrate nella tabella di Figura 9.21, anche se la vera formattazione è differente ed è correlata alla gerarchia delle classi Java.

Un tipo di azione permette l'accesso al file; l'azione può specificare un file o una directory specifici, l'insieme di tutti file in una directory, o l'insieme di tutti i file e le directory

| URL               | Chi firma   | Oggetto             | Azione              |
|-------------------|-------------|---------------------|---------------------|
| www.taxprep.com   | TaxPreparer | /usr/susan/1040.xls | Read                |
| *                 |             | /usr/tmp/-          | Read, Write         |
| www.microsoft.com | Microsoft   | /usr/susan/Office/- | Read, Write, Delete |

Figura 9.21 Alcuni esempi di protezione che si possono specificare con JDK 1.2.

ricorsivamente contenuti in una data directory. Le tre righe di Figura 9.21 corrispondono a questi casi: nella prima linea, l'utente, Susan, ha organizzato il suo file dei permessi in modo che le applet che arrivano dalla macchina della ditta del suo commercialista, la Tax Preparer, [www.taxprep.com](http://www.taxprep.com), e firmate da tale azienda, abbiano diritto di lettura sui suoi dati relativi alle tasse, presenti nel file *1040.xls*. Questo è l'unico file che possono leggere e nessun'altra applet può leggere questo file; inoltre tutte le applet, qualunque sia la loro origine e indipendentemente dal fatto che siano firmate o meno, possono leggere e scrivere file in */usr/tmp*.

Oltre a ciò, Susan si fida anche di Microsoft, abbastanza da permettere alle applet provenienti dal suo sito e firmate da Microsoft di scrivere, leggere e cancellare tutti i file presenti nella directory *Office* e nelle sue sottodirectory, ad esempio, per correggere bachi ed installare nuove versioni del software. Per verificare la firma, Susan deve già possedere la chiave pubblica necessaria, o deve acquisirla dinamicamente, ad esempio tramite un certificato firmato da un'azienda di cui si fida e di cui già possiede la chiave pubblica.

I file non sono le sole risorse che si possono proteggere; si può proteggere anche l'accesso alla rete: in questo caso gli oggetti sono porte specifiche su macchine specifiche. Un computer è identificato da un indirizzo IP o un nome del DNS: le porte di una certa macchina sono specificate da un intervallo di numeri. Le azioni possibili comprendono richiedere una connessione ad un computer remoto ed accettare connessioni generate da un computer remoto; in questo modo, un'applet può ottenere l'accesso alla rete, ma limitato alla connessione con quei computer esplicitamente nominati nella lista dei permessi. Le applet possono caricare dinamicamente del codice (classi) supplementare a seconda delle loro necessità, ma la classe addetta al caricamento e fornita dall'utente può controllare precisamente su quali macchine tali classi abbiano avuto origine. Naturalmente, esistono numerose altre funzionalità di sicurezza.

## 9.6 Meccanismi di protezione

Nelle sezioni precedenti abbiamo illustrato molti problemi potenziali, alcuni dei quali tecnici, altri no. Nelle sezioni successive ci soffermeremo su alcuni dei dettagli tecnici utilizzati dai sistemi operativi per proteggere file ed altro. Tutte queste tecniche pongono una chiara distinzione tra politiche (i dati di chi, e da chi devono essere protetti) e meccanismi (come il sistema applica le politiche); la separazione tra politiche e meccanismi è discussa in Sandhu (1993) e noi metteremo a fuoco i meccanismi, non le politiche.

In alcuni sistemi la protezione è realizzata tramite un programma, chiamato **monitor degli indirizzi**; ogni volta che vi è un tentativo di accedere ad una risorsa potenzialmente protetta, il sistema dapprima chiede al monitor degli indirizzi di controllarne la legalità; il monitor degli indirizzi quindi controlla la sue tabelle (di politica) e prende una decisione. Qui sotto descriviamo l'ambiente in cui agisce un monitor degli indirizzi.

### 9.6.1 Domini di protezione

Un sistema contiene molti "oggetti" che hanno bisogno di essere protetti; questi oggetti possono essere sia hardware (ad esempio CPU, segmenti di memoria, terminali, drive per dischi o stampanti), sia software (ad esempio processi, file, basi di dati o semafori).

Ogni oggetto ha un nome unico con cui è riconosciuto e un insieme finito di operazioni che si possono eseguire su esso. Le operazioni di **read** e **write** sono appropriate per un file; **up** e **down** hanno senso per un semaforo.

È ovvio che sia necessario un modo per impedire ai processi di accedere ad oggetti

a cui non sono autorizzati; inoltre il meccanismo deve anche far sì che i processi possano essere limitati ad un sottoinsieme delle operazioni legali, quando ciò è necessario. Ad esempio un processo *A* può essere autorizzato a leggere, ma non a scrivere, un file *F*.

Per fornire un modo di discutere i differenti meccanismi di protezione, è conveniente introdurre il concetto di dominio. Un **dominio** è un insieme di coppie (oggetto, diritti). Ogni coppia specifica un oggetto e qualche sottoinsieme di operazioni che si possono eseguire su di esso: un **diritto**, in questo contesto, significa il permesso di eseguire una delle operazioni; spesso un dominio corrisponde ad un singolo utente, e ci dice cosa l'utente possa o non possa fare, ma un dominio può anche essere più generico di un solo utente.

La Figura 9.22 mostra tre domini, con gli oggetti in ogni dominio e i diritti (Read, Write, eXecute) disponibili su ogni oggetto. Si noti che *Printer1* è in due domini contemporaneamente e, sebbene non mostrato nell'esempio, per lo stesso oggetto è possibile essere presente in più domini, con diritti *differenti* in ogni dominio.

In ogni istante, ogni processo è eseguito in qualche dominio di protezione, in altre parole vi è un qualche insieme di oggetti a cui esso può accedere e, per ognuno di questi, esso ha dei diritti. I processi possono anche cambiare dominio durante l'esecuzione; le regole per cambiare dominio sono altamente dipendenti dal sistema.

Per rendere più concreta l'idea di dominio di protezione si pensi a UNIX: qui il dominio di un processo è definito dai suoi UID e GID; data ogni combinazione (UID, GID), è possibile fare una lista completa di tutti gli oggetti (file, comprese le periferiche di I/O rappresentate da file speciali, eccetera) a cui si può accedere, ed indicare se essi possono essere visitati per lettura, scrittura o esecuzione. Due processi con la stessa combinazione (UID, GID) accederanno esattamente allo stesso insieme di oggetti, mentre processi con differenti valori (UID, GID) accederanno a insiemi differenti di file, sebbene in molti casi vi saranno considerevoli sovrapposizioni.

Inoltre ogni processo in UNIX ha due metà: la parte utente e la parte kernel; quando un processo effettua una chiamata di sistema si sposta dalla parte utente alla parte kernel. La parte kernel ha accesso a un insieme di oggetti differente da quello della parte utente: ad esempio, il kernel può accedere a tutte le pagine nella memoria fisica, a tutto il disco e a tutte le risorse protette; così una chiamata di sistema provoca un cambio di dominio.

Quando un processo effettua una exec su un file avendo a 1 il bit SETUID o SETGID, assume effettivamente un nuovo UID o GID; con una combinazione differente (UID, GID), esso ha un insieme differente di file e di operazioni disponibili. Si ha un cambio di dominio anche quando si esegue un programma con SETUID o SETGID, perché i diritti disponibili cambiano.

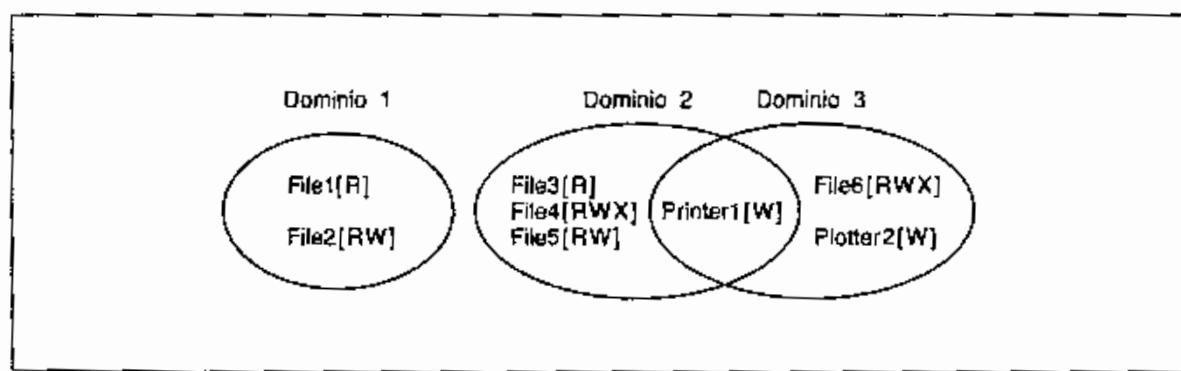


Figura 9.22 Tre domini di protezione.

|         |   | Oggetto |                      |         |                                    |                      |                                    |           |           |
|---------|---|---------|----------------------|---------|------------------------------------|----------------------|------------------------------------|-----------|-----------|
|         |   | File1   | File2                | File3   | File4                              | File5                | File6                              | Printer1  | Plotter2  |
| Dominio | 1 | Lettura | Lettura<br>Scrittura |         |                                    |                      |                                    |           |           |
|         | 2 |         |                      | Lettura | Lettura<br>Scrittura<br>Esecuzione | Lettura<br>Scrittura |                                    | Scrittura |           |
|         | 3 |         |                      |         |                                    |                      | Lettura<br>Scrittura<br>Esecuzione | Scrittura | Scrittura |

Figura 9.23 Una matrice di protezione.

Un problema importante è come il sistema tenga traccia di quali oggetti appartengano a quali domini; concettualmente, si può immaginare una grande matrice, dove le righe sono i domini e le colonne gli oggetti: ogni elemento mostra, se ve ne sono, i diritti che il dominio contiene per l'oggetto. La matrice di Figura 9.22 è mostrata in Figura 9.23; data questa matrice ed il numero di dominio corrente, il sistema può stabilire se concedere un accesso ad un dato oggetto, in un modo particolare e da un dominio specifico.

Lo stesso cambio di dominio può essere facilmente incluso nel modello a matrice, pensando che il dominio stesso è un oggetto, con l'operazione `enter`. La Figura 9.24 mostra la matrice di Figura 9.23 di nuovo, in questo caso con i tre domini sotto forma di oggetti; i processi del dominio 1 possono cambiare al dominio 2, ma una volta fatto ciò, non possono tornare indietro: questa situazione modella il modo di agire di un programma SETUID in UNIX. Nessun altro cambio di dominio è consentito nell'esempio.

|         |   | Oggetto |                      |         |                                    |                      |                                    |           |           |          |          |          |
|---------|---|---------|----------------------|---------|------------------------------------|----------------------|------------------------------------|-----------|-----------|----------|----------|----------|
|         |   | File1   | File2                | File3   | File4                              | File5                | File6                              | Printer1  | Plotter2  | Dominio1 | Dominio2 | Dominio3 |
| Dominio | 1 | Lettura | Lettura<br>Scrittura |         |                                    |                      |                                    |           |           |          |          | Enter    |
|         | 2 |         |                      | Lettura | Lettura<br>Scrittura<br>Esecuzione | Lettura<br>Scrittura |                                    | Scrittura |           |          |          |          |
|         | 3 |         |                      |         |                                    |                      | Lettura<br>Scrittura<br>Esecuzione | Scrittura | Scrittura |          |          |          |

Figura 9.24 Una matrice di protezione con domini sotto forma di oggetti.

## 9.6.2 Liste di controllo degli accessi

In pratica, la memorizzazione della matrice di Figura 9.24 è realizzata raramente perché occupa spazio ed è sparsa; la maggior parte dei domini non hanno alcun accesso alla maggioranza degli oggetti, per cui memorizzare una grossa matrice praticamente vuota significa sprecare spazio su disco. Due metodi pratici, comunque, sono la memorizzazione della matrice per righe o per colonne, memorizzando solo gli elementi non vuoti: i due approcci sono sorprendentemente differenti. In questa sezione si considererà la memorizzazione per colonne, nella prossima quella per righe.

La prima tecnica consiste nell'associare ad ogni oggetto una lista ordinata che contiene tutti i domini che possono accedere all'oggetto e come; questa lista è detta **lista di controllo degli accessi** (access control list) o **ACL** ed è mostrata in Figura 9.25: vediamo tre processi, ognuno dei quali appartiene ad un differente dominio; A, B, e C, e tre file F1, F2 e F3. Per semplicità supporremo che ad ogni dominio corrisponda esattamente un utente; in questo caso gli utenti A, B, e C. Spesso nella letteratura che tratta di sicurezza, gli utenti sono chiamati **soggetti** o **principal**, per porli in contrasto con ciò che è posseduto, gli **oggetti**, come i file.

Ogni file ha un'ACL associata ad esso; il file F1 ha due elementi nella sua ACL (separate da punto e virgola). Il primo elemento dice che ogni processo posseduto dall'utente A può leggere e scrivere il file; il secondo dice che ogni processo posseduto dall'utente B può leggere il file; sono vietati tutti gli altri accessi da parte di questi utenti e tutti gli accessi da parte di tutti gli altri utenti. Notiamo che i diritti sono concessi all'utente, non al processo. Finché il sistema di protezione funziona, ogni processo posseduto dall'utente A può leggere e scrivere il file F1; non ha importanza se vi siano 1 o 100 di questi processi, quello che conta è il proprietario, non l'ID del processo.

Il file F2 ha tre entry nella sua ACL: A, B e C possono leggere il file ed inoltre B può anche scriverlo; non sono consentiti altri accessi. Apparentemente il file F3 è un programma eseguibile, poiché B e C possono entrambi leggerlo ed eseguirlo; B può anche scriverlo.

Questo esempio mostra la forma più semplice di protezione con le ACL, anche se spesso nella pratica, sono utilizzati sistemi più sofisticati. Per cominciare, abbiamo mostrato

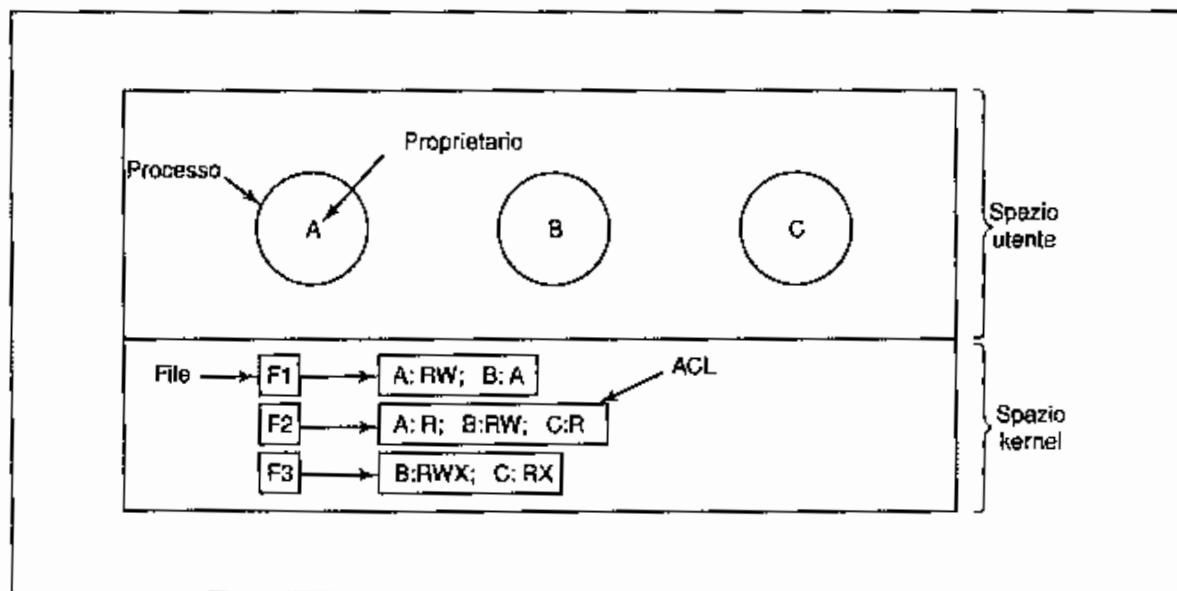


Figura 9.25 Uso delle liste di controllo degli accessi per gestire l'accesso ai file.

solo tre diritti: **read**, **write** ed **execute**, ma vi possono essere altri diritti. Alcuni di essi possono essere generici, cioè applicabili a tutti gli oggetti, altri possono essere specifici per certi oggetti; esempi di diritti generici possono essere **destroy object** e **copy object** (distruggi e copia oggetto). Questi possono essere applicati a qualunque oggetto, di qualunque tipo. Al contrario, diritti per oggetti specifici potrebbero essere **append message** (aggiungi messaggio) per un oggetto **mailbox** e **sort alphabetically** (ordina alfabeticamente) per l'oggetto **directory**.

Le nostre ACL sono state descritte per utenti singoli, ma molti sistemi supportano il concetto di **gruppo** di utenti: i gruppi hanno un nome e possono essere inclusi nelle ACL. Sono possibili due variazioni nella semantica dei gruppi: in alcuni sistemi, ogni processo ha un user ID (UID) e un group ID (GID); in tali sistemi, l'entry di un'ACL contiene elementi della forma

```
UID1, GID1: diritti1; UID2, GID2: diritti2; ...
```

In tali condizioni, quando si richiede un accesso ad un oggetto, si controllano UID e GID del richiedente: se sono presenti nell'ACL, sono disponibili i diritti che vi compaiono; se invece la combinazione (UID,GID) non è presente nella lista, l'accesso non è consentito.

Utilizzare i gruppi in questo modo, introduce il concetto di **ruolo**. Consideriamo una installazione in cui Tina è l'amministratore di sistema, e dunque nel gruppo **sysadm**; tuttavia supponiamo che l'azienda abbia anche dei club per impiegati, e Tina sia un membro del club degli appassionati dei piccioni. I membri di tale club appartengono al gruppo **pigfan** ed hanno accesso ai computer dell'azienda per gestire i loro database. Una parte dell'ACL potrebbe essere mostrata in Figura 9.26.

Se Tina cerca di accedere ad uno di questi file, il risultato dipende dal gruppo di appartenenza della login con cui ha avuto accesso; quando effettua il login, il sistema può chiederle quale dei suoi gruppi stia usando attualmente, o vi potrebbero anche essere differenti coppie login-password per tenerli separati. Il punto saliente dello schema è evitare che Tina abbia accesso al file delle password quando è nelle vesti di membro del club, ma lo possa solamente fare in qualità di amministratore di sistema.

In alcuni casi, l'utente potrebbe avere accesso a certi file in modo indipendente dal gruppo con cui ha effettuato il login; si può affrontare questo caso, introducendo delle **wildcard**, che significa chiunque. Ad esempio, l'entry

```
tina, *:RW
```

per il file delle password, darebbe l'accesso a Tina, indipendentemente dal gruppo con cui ha avuto accesso.

Ancora un'altra possibilità si ha se l'accesso è consentito quando l'utente appartiene ad uno qualsiasi dei gruppi che hanno certi diritti di sicuro accesso; in questo caso, un utente che appartenga a più gruppi non deve specificare quale gruppo utilizzare al momento del login: tutti contano sempre. Uno svantaggio di questo approccio è che fornisce meno encapsulazione: Tina può editare il file delle password durante una riunione del suo club.

| File              | Access control list                     |
|-------------------|-----------------------------------------|
| Password          | tina, sysadm: RW                        |
| Dati_dei_piccioni | bill, pigfan: RW; tina, pigfan: RW; ... |

Figura 9.26 Due liste di controllo degli accessi.

L'uso di gruppi e wildcard introduce la possibilità di bloccare selettivamente uno specifico utente dall'accedere ad un file; ad esempio, l'entry

```
virgil,*:(none); *,*:RW
```

dà al mondo intero l'accesso in lettura e scrittura al file, tranne che a Virgil. Questo succede perché le entry sono esaminate in ordine, e la prima che funziona, viene scelta, mentre le entry successive non sono nemmeno esaminate: si trova una corrispondenza per Virgil nella prima entry; si trovano anche i diritti di accesso (nessuno); vengono applicati e la ricerca termina a questo punto. Il fatto che al resto del mondo venga dato accesso, non viene nemmeno preso in considerazione.

L'altro modo per gestire i gruppi è non avere le entry delle ACL formate da coppie (UID, GID), ma avere come entry un UID o un GID; ad esempio, un entry per il file *Dati\_dei\_piccioni*, potrebbe essere

```
daria: RW; piero: RW; pigfan: RW
```

che significa che Daria, Piero e tutti i membri del gruppo *pigfan* hanno diritto di lettura e scrittura su file.

Talvolta accade che un utente o un gruppo abbia permessi su un file, che poi il proprietario del file vuole revocare; con le ACL è relativamente semplice revocare un accesso che si era precedentemente dato: tutto quello che si deve fare è editare il file delle ACL per fare la modifica. Tuttavia se le ACL sono controllate solo quando il file è aperto, la modifica avrà probabilmente effetto a partire dalle chiamate a open successive. Ogni file che è già aperto continuerà ad avere i diritti di quando è stato aperto, anche se l'utente non è più autorizzato ad accedere al file.

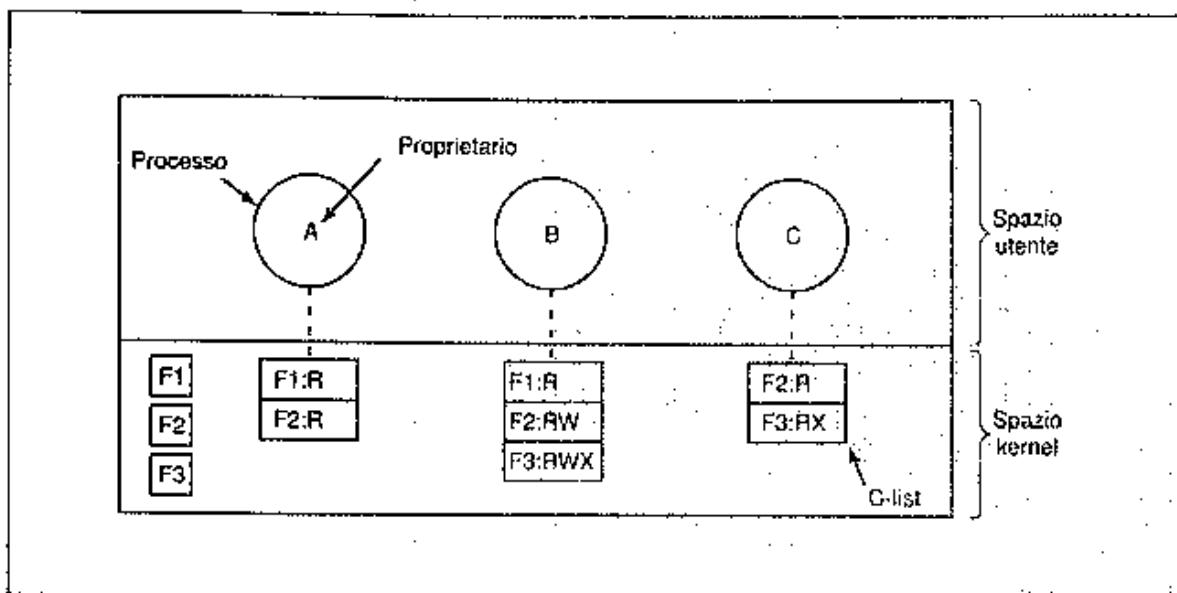
### 9.6.3 Capability

L'altro modo per ridurre la matrice di Figura 9.24 è per righe: quando si usa questo metodo, ad ogni processo è associata una lista di oggetti a cui si può accedere, insieme con un'indicazione di quali operazioni sono permesse su ognuno, in altre parole il suo dominio. Questa lista è detta **capability list** (lista di capacità) o C-List e i singoli elementi in essa sono chiamati **capability** (Dennis e Van Horn, 1966; Fabry, 1974). In Figura 9.27 viene mostrato un insieme di tre processi e la loro lista di capability.

Ogni capability garantisce al proprietario certi diritti su certi oggetti; in Figura 9.27, il processo dell'utente *A* può leggere i file *F1* e *F2*, ad esempio. Di solito, una capability si compone di un identificatore di file (o più genericamente, un oggetto) ed una mappa di bit per i vari diritti; in un sistema UNIX, l'identificatore di file sarà probabilmente l'i-node. Le liste di capability sono esse stesse degli oggetti, e possono essere puntate da altre liste di capability, facilitando in questo modo la condivisione dei sottodomini.

È ovvio che le liste di capability devono essere protette da manomissioni degli utenti, e sono noti tre metodi per fare ciò: il primo metodo richiede un architettura etichettata, un progetto hardware in cui ogni parola della memoria ha un bit extra (o tag), che dice se la parola contiene o meno una capability. Il bit tag non è utilizzato dall'aritmetica, dai confronti o da altre istruzioni normali, e può essere modificato solo da programmi che agiscono in modo kernel (ad esempio, il sistema operativo). Sono state costruite macchine con architettura etichettata e sono in grado di lavorare bene (Feustal, 1972); un esempio popolare è l'IBM AS/400.

Il secondo metodo è mantenere la C-list all'interno del sistema operativo; le capability



**Figura 9.27** Quando sono utilizzate le capability, ogni processo ha una lista di capability.

sono poi indirizzate tramite la loro posizione nella lista di capability; ad esempio, un processo può dire: "Leggi 1K dal file puntato dalla capability 2". Questa forma di indirizzamento è simile all'uso dei descrittori del file in UNIX, ed Hydra (Wulf et al., 1974) lavorava in tale modo.

Il terzo metodo consiste nel mantenere la C-list nello spazio utente, ma crittografare ogni capability in modo da evitare manipolazioni degli utenti. Questo approccio è particolarmente adatto per sistemi distribuiti e funziona come segue: quando un processo client manda un messaggio ad un server remoto, ad esempio un file server, per creare un oggetto per esso, il server crea l'oggetto e genera un numero lungo casuale, il check field, associato ad esso. Uno slot nella tabella del file server è riservato per l'oggetto, e il check field è ivi memorizzato, assieme agli indirizzi dei blocchi disco, eccetera. In termini UNIX, il check field viene memorizzato sul server nell'i-node; non è mandato indietro all'utente, e non viene mai messo in rete. Il server quindi genera e restituisce all'utente una capability, nella forma mostrata in Figura 9.28.

La capability restituita all'utente contiene l'identificatore del server, il numero dell'oggetto (l'indice nelle tabelle del server, essenzialmente, il numero di i-node) e i diritti, memorizzati in una mappa di bit. Per un oggetto appena creato, i bit dei diritti sono tutti a 1; l'ultimo campo consiste di una concatenazione di oggetto, diritti e check field, passati attraverso una funzione unidirezionale,  $f$ , crittograficamente sicura, simile a quelle già discusse in precedenza.

Quando un utente vuole accedere all'oggetto, questi manda la capability al server, come parte della richiesta; il server estrae quindi il numero dell'oggetto, per recuperare l'indice nelle sue tabelle e trovare l'oggetto; calcola poi  $f(Oggetto, Diritti, Check)$  prendendo i primi due parametri dalla stessa capability ed il terzo dalle sue tabelle. Se il risultato coincide con il quarto campo della capability, la richiesta è accettata, altrimenti viene

| Server | Oggetto | Diritti | $f(Oggetto, Diritti, Check)$ |
|--------|---------|---------|------------------------------|
|--------|---------|---------|------------------------------|

**Figura 9.28** Una capability protetta con crittografia.

rifiutata. Se un utente cerca di accedere all'oggetto di qualcun altro, non sarà in grado di generare il quarto campo in modo corretto perché non conosce il check field e dunque la richiesta verrà rifiutata.

Un utente può chiedere al server di produrre una capability più debole, ad esempio, per il solo accesso in lettura: dapprima il server verifica che la capability sia valida; se lo è, calcola  $f(Oggetto, NuoviDiritti, Check)$  e genera una nuova capability, mettendo questo valore nel quarto campo. Notiamo che viene utilizzato il valore originale *Check*, in quanto altre capability in sospeso dipendono da esso.

Al processo richiedente viene restituita questa nuova capability; l'utente può quindi darla ad un amico, semplicemente allegandola ad un messaggio. Se tale amico porta a 1 i bit dei diritti, che dovrebbero essere a 0, il server lo rileverà quando viene utilizzata la capability, perché il valore di *f* non corrisponderà al valore falsificato dei diritti. Siccome l'amico non conosce il vero check field, non può generare una capability che corrisponda ai bit dei diritti falsificati. Questo schema è stato sviluppato per il sistema Amoeba ed è molto utilizzato (Tanenbaum et al., 1990).

Oltre ai diritti specifici dipendenti dall'oggetto, come lettura ed esecuzione, le capability di solito hanno diritti generici che sono applicabili a tutti gli oggetti. Esempi di diritti generici sono:

1. **Copia capability:** crea una nuova capability per lo stesso oggetto.
2. **Copia oggetto:** crea un oggetto duplicato con una nuova capability.
3. **Cancella capability:** cancella un elemento dalla C-list; l'oggetto resta inalterato.
4. **Distruzione oggetto:** rimuove definitivamente un oggetto e una capability.

Un'ultima osservazione che vale la pena fare sui sistemi a capability è che revocare i diritti di accesso nei confronti di un oggetto è notevolmente difficile nella versione gestita dal kernel. Per il sistema è arduo rintracciare tutte le capability in sospeso per ogni oggetto per poi ritirarle, poiché esse possono essere memorizzate in C-list sparse ovunque nel disco. Un approccio è quello di far sì che ogni capability punti a un oggetto indiretto piuttosto che all'oggetto stesso; dato che l'oggetto indiretto punta all'oggetto reale, il sistema può sempre interrompere la comunicazione invalidando così le capability. (In seguito, presentando al sistema una capability all'oggetto indiretto, l'utente scoprirà che l'oggetto indiretto punta a un oggetto nullo).

Un altro metodo più semplice per ottenere la revoca è lo schema usato in Amoeba: tutto quello che è necessario fare è cambiare il check field memorizzato nell'oggetto, così, in un colpo solo, tutte le capability esistenti vengono invalidate. Tuttavia nessuno degli schemi permette una revoca selettiva, cioè, togliere il permesso a Gino e a nessun altro. Questo difetto è generalmente riconosciuto come un problema per tutti i sistemi a capability.

Un altro problema generale è essere sicuri che il proprietario di una capability legale non ne dia una copia a 1000 suoi cari amici: il kernel che gestisce le capability, come Hydra, risolve il problema, ma questa soluzione non funziona bene nei sistemi distribuiti come Amoeba.

D'altro canto, le capability risolvono in maniera molto elegante il problema di mettere il codice mobile in sandbox: quando viene eseguito un programma sconosciuto, gli viene assegnata una lista di capability che contiene solamente quelle capability che il proprietario della macchina vuole dargli, come la possibilità di scrivere sullo schermo, leggere e scrivere file in una directory temporanea, creata appositamente. Se il codice mobile viene inserito nel suo processo con queste limitate capability, non sarà in grado di accedere alle altre risorse del sistema e sarà effettivamente confinato nel sandbox, senza la necessità di modificare il suo codice o di eseguirlo in maniera interpretativa. Eseguire codice con

il minimo necessario dei diritti di accesso è noto come il **principio del minimo privilegio** e rappresenta la linea guida per la produzione di sistemi sicuri.

Riassumendo brevemente, ACL e capability hanno in qualche modo delle proprietà complementari: le capability sono molto efficienti, perché se un processo dice "Apri il file puntato dalla capability 3", non c'è bisogno di controlli; con le ACL potrebbe essere necessaria una (potenzialmente lunga) ricerca delle ACL. Se non sono supportati i gruppi, allora garantire a tutti l'accesso in lettura, richiede l'elenco di tutti gli utenti nelle ACL; le capability permettono anche di incapsulare facilmente un processo, laddove le ACL non lo permettono. D'altro canto, le ACL permettono di revocare in maniera selettiva i diritti, mentre le capability no; infine, se un oggetto viene rimosso, ma non le capability, oppure le capability sono rimosse ma non l'oggetto, sorgono dei problemi, mentre le ACL non soffrono questo problema.

## 9.7 Sistemi fidati

Molto di questo capitolo è stato dedicato al fatto che virtualmente tutti i moderni sistemi di computer sono dei colabrodo: le stime dei danni a livello mondiale, causati da virus e problemi simili, sono superiori a 1000 miliardi di dollari all'anno impiegati nello sforzo inutile di rimediare a problemi, ricostruire dati danneggiati, eccetera, per non parlare delle opportunità di affari andate perse. Una persona ingenua potrebbe logicamente porsi due domande a proposito di ciò:

1. È possibile costruire un sistema di computer sicuro?
2. Se è così, perché non lo si fa?

La risposta alla prima domanda è fondamentalmente sì: come costruire un sistema sicuro è noto da decenni. MULTICS, ad esempio, progettato negli anni '60, pone la sicurezza come uno dei suoi obiettivi principali e la realizza abbastanza bene.

Perché non si costruiscono sistemi sicuri è più complicato, ma deriva da due ragioni fondamentali: primo, gli attuali sistemi non sono sicuri, ma gli utenti non vogliono rifiutarli. Se Microsoft annunciasse che assieme a Windows ha creato un nuovo prodotto, SecureOS, che si garantisce che sarà immune ai virus, ma non eseguirà le applicazioni di Windows, è molto poco probabile che ogni persona e ogni azienda getti via Windows come una patata bollente, per comprare immediatamente il nuovo sistema.

La seconda questione è molto più sottile: l'unico modo per costruire un sistema sicuro è renderlo semplice. Le funzionalità sono nemiche della sicurezza, ed i progettisti di sistema ritengono (giusto o sbagliato che sia) che quello che gli utenti vogliono siano le funzionalità; ma più funzionalità, significano più complessità, più codice, più bug e più errori di sicurezza.

Ecco due semplici esempi. Il primo sistema di e-mail mandava messaggi in formato testo ASCII ed erano completamente sicuri, perché non c'è nulla di dannoso che un messaggio ASCII in arrivo possa fare al sistema. In seguito la gente ebbe l'idea di estendere e-mail in modo che potesse includere altri tipi di documenti, ad esempio, file in Word, che possono contenere macro. Leggere tali documenti significa eseguire il programma di qualcun altro sul proprio computer ed indipendentemente da quanto sandboxing è utilizzato, eseguire un programma sconosciuto sul proprio computer è decisamente più pericoloso che guardare un testo ASCII. Gli utenti chiesero la possibilità di cambiare il programma e-mail da passivo ad attivo? Probabilmente no, ma i progettisti pensarono

che fosse una buona idea, senza preoccuparsi troppo delle implicazioni a livello di sicurezza.

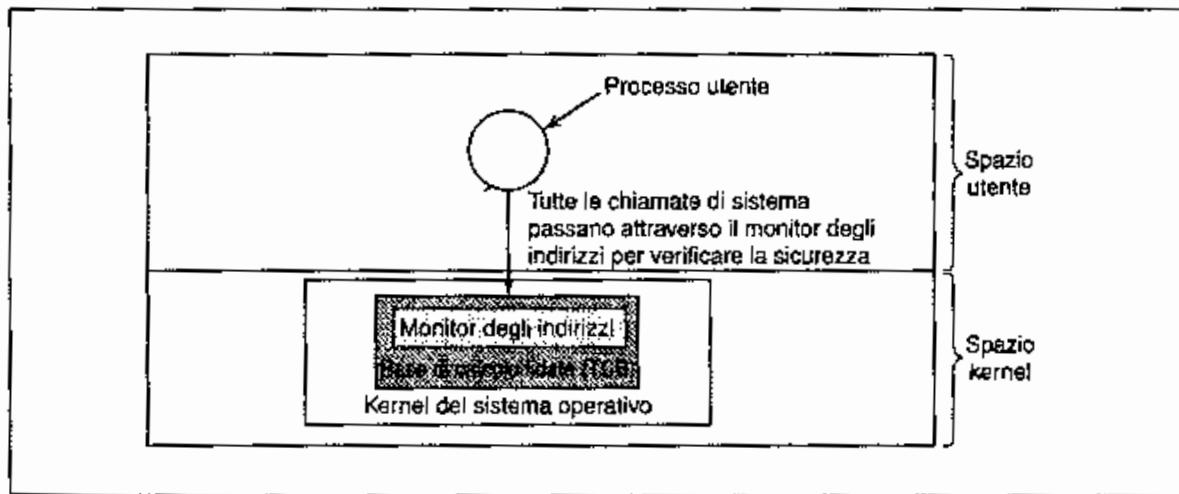
Il secondo esempio è la stessa cosa, ma per pagine Web. Quando il Web consisteva di pagine HTML passive, non poneva grandi problemi di sicurezza (sebbene l'HTML illegale possa causare un attacco tramite buffer overflow), ma, ora che molte pagine Web contengono programmi (applet) che l'utente deve eseguire per visualizzare il contenuto, si presenta una falla nella sicurezza dopo l'altra: non appena ne viene tappata una, ne compare un'altra. Quando il Web era completamente statico, gli utenti erano in armi per chiedere contenuti dinamici? No, a quanto ricorda l'autore, ma la loro introduzione portò con sé una raffica di problemi alla sicurezza. È come se il vice presidente responsabile di dire no stesse dormendo al volante!

In realtà esistono alcune organizzazioni che pensano che una buona sicurezza sia più importante di nuove graziose funzionalità, ed il primo esempio sono i militari. Nelle sezioni che seguono indagheremo alcune delle questioni implicate, che possono essere riassunte in una frase: per costruire un sistema sicuro, occorre avere al centro del sistema operativo un modello di sicurezza abbastanza semplice da poter essere realmente compreso dai progettisti, e resistere a tutte le pressioni per distaccarsi da esso, al fine di aggiungere nuove funzionalità.

### 9.7.1 Basi di calcolo fidate (TCB)

Nel mondo della sicurezza, spesso la gente parla di sistemi fidati piuttosto che di sistemi sicuri; questi sono sistemi che hanno stabilito formalmente dei requisiti di sicurezza e li rispettano. Alla base di ogni sistema fidato vi è un minimo TCB (Trusted Computing Base, base di calcolo fidata), composto dall'hardware e dal software necessario per fare rispettare le regole di sicurezza. Se il TCB sta lavorando secondo le specifiche, la sicurezza del sistema non può essere compromessa, indipendentemente da altri errori.

Il TCB di solito è composto in maggior parte da hardware (eccetto i dispositivi di I/O che non influiscono sulla sicurezza), da una porzione del kernel del sistema operativo e dalla maggior parte o da tutti i programmi dell'utente che hanno i diritti di superuser (ad esempio, il programma SETUID di root in UNIX). Le funzioni del sistema operativo che devono essere parte del TCB comprendono la creazione dei processi, il cambio di processo, la gestione delle mappe di memoria, la gestione di una parte del meccanismo dei file



**Figura 9.29** Un monitor degli indirizzi.

e di I/O. In un progetto sicuro, spesso il TCB sarà abbastanza separato dal resto del sistema operativo, al fine di minimizzare la sua dimensione e verificarne la correttezza.

Una parte importante del TCB è il monitor degli indirizzi, come mostrato nella Figura 9.29. Il monitor degli indirizzi accetta tutte le chiamate di sistema che coinvolgono la sicurezza, come aprire file, e decide se debbano essere eseguiti o meno; in questo modo il monitor degli indirizzi fa sì che tutte le decisioni concernenti la sicurezza siano in un solo luogo e non vi siano possibilità di aggirarlo. La maggior parte dei sistemi operativi non sono progettati in questo modo, e questa è una parte del motivo per cui sono così insicuri.

## 9.7.2 Modelli formali di sistemi sicuri

Le matrici di protezione, come quelle di Figura 9.23, non sono statiche: variano frequentemente quando si creano nuovi oggetti, si distruggono i vecchi, e quando i proprietari decidono di estendere o restringere l'insieme degli utenti che hanno diritti sui loro oggetti. La modellazione dei sistemi di protezione in cui la matrice di protezione è costantemente in via di modifica è stato l'oggetto di molte ricerche, e nella parte restante di questo paragrafo si descriveranno brevemente alcuni di questi lavori.

Anni fa, Harrison ed al. (1976) hanno identificato sei operazioni primitive sulla matrice di protezione che si possono usare come base per modellare qualunque sistema di protezione. Queste operazioni sono: `create object`, `delete object`, `create domain`, `delete domain`, `insert right` e `remove right`. Le ultime due primitive inseriscono e tolgono i diritti dagli elementi specifici di una matrice, in modo, ad esempio, da accordare al dominio 1 il permesso di leggere il *File6*.

Queste sei primitive si possono unire in comandi di protezione: sono questi comandi di protezione quelli che i programmi utenti possono eseguire per modificare la matrice. Ad esempio, il sistema potrebbe avere un comando per creare un nuovo file, che controlla se il file già esiste e, in caso contrario, crea un nuovo oggetto e dà al proprietario tutti i diritti su di esso. Ci potrebbe anche essere un comando per permettere al proprietario di accordare il permesso per leggere il file a chiunque nel sistema, inserendo il diritto di lettura "read" nel nuovo elemento del file in ogni dominio.

Ad ogni istante la matrice determina cosa può fare un processo in ogni dominio, non cosa è autorizzato a fare; la matrice descrive ciò che il sistema deve controllare: l'autorizzazione ha a che fare con la politica di gestione. Come esempio di questa distinzione, si consideri il semplice sistema di Figura 9.30 in cui i domini corrispondono agli utenti. Nella Figura 9.30(a) si vede la politica di protezione adottata: Enrico può leggere e scrivere la *mailbox7*, Roberto può leggere e scrivere *Secret*, e tutti e tre gli utenti possono leggere ed eseguire *Compilatore*.

Ora si immagini che Roberto sia molto intelligente ed abbia trovato un modo per lanciare dei comandi che modificano la matrice come in Figura 9.30(b): a questo punto ha avuto accesso alla *mailbox7* senza esserne autorizzato; se provasse a leggere *mailbox7*, il sistema operativo eseguirebbe la sua richiesta perché non sa che lo stato di Figura 9.30(b) non è autorizzato.

Dovrebbe essere chiaro che l'insieme di tutte le possibili matrici può essere diviso in due insiemi disgiunti: l'insieme di tutti gli stati autorizzati e l'insieme di tutti gli stati non autorizzati. Una domanda che ha dato origine a molti studi teorici è la seguente: "Dato uno stato iniziale autorizzato e un insieme di comandi si può dimostrare che il sistema non raggiungerà mai uno stato non autorizzato?"

In effetti la domanda è se il meccanismo disponibile (i comandi di protezione) è adeguato per imporre qualche politica di protezione. Data questa politica, uno stato iniziale

The diagram shows two tables representing security matrices. Both tables have columns for 'Compilatore' (Eric, Enrico, Roberto), 'Oggetti' (Mailbox 7, Secret), and rows for 'Lettura' and 'Esecuzione'. In table (a), all access rights are granted. In table (b), access to 'Secret' objects is denied for all users.

|      |         |            | Oggetti     |           |           |
|------|---------|------------|-------------|-----------|-----------|
|      |         |            | Compilatore | Mailbox 7 | Secret    |
| Eric | Lettura | Esecuzione |             |           |           |
|      | Enrico  | Lettura    | Esecuzione  | Lettura   | Scrittura |
|      | Roberto | Lettura    | Esecuzione  |           | Lettura   |

(a)

|      |         |            | Oggetti     |           |           |
|------|---------|------------|-------------|-----------|-----------|
|      |         |            | Compilatore | Mailbox 7 | Secret    |
| Eric | Lettura | Esecuzione |             |           |           |
|      | Enrico  | Lettura    | Esecuzione  | Lettura   | Scrittura |
|      | Roberto | Lettura    | Esecuzione  | Lettura   |           |

(b)

Figura 9.30 (a) Uno stato autorizzato, (b) Uno stato non autorizzato.

della matrice, ed un insieme di comandi per modificare la matrice, quello che ci piacerebbe è un modo per provare che il sistema è sicuro. Una tale dimostrazione risulta abbastanza difficile da acquisire, e dunque molti sistemi general purpose sono teoricamente non sicuri. Harrison et al. (1976) provarono che, nel caso di una configurazione arbitraria, per una protezione del sistema arbitraria, la sicurezza è teoricamente indecidibile; tuttavia, specificato un sistema, può essere possibile provare se il sistema può passare da un stato autorizzato ad uno stato non autorizzato. Per maggiori informazioni, si veda Landwehr (1981).

### 9.7.3 Sicurezza multilivello

La maggior parte dei sistemi operativi permette a singoli utenti di determinare chi possa leggere e scrivere i loro file ed i loro oggetti: questa politica è detta **controllo discrezionale degli accessi**. In molti ambienti questo modello funziona bene, ma vi sono altri ambienti in cui è richiesta una sicurezza più stretta, come l'ambiente militare, gli uffici brevetti e gli ospedali. In questi ultimi ambienti, l'organizzazione ha stabilito regole su chi può vedere e cosa, e non possono essere modificate da soldati, avvocati o dottori, o almeno non senza aver ottenuto dei permessi speciali dal capo. Questi ambienti necessitano di **controllo dell'accesso obbligatorio** per assicurarsi che le politiche di sicurezza stabilite siano rispettate dal sistema, oltre al controllo discrezionale degli accessi standard. Quello che fanno i controlli dell'accesso obbligatorio è regolare il flusso di informazioni, per essere sicuri che non ve ne sia una perdita in modi che non dovrebbero verificarsi.

#### Il modello Bell-La Padula

Il modello di sicurezza multilivello più ampiamente utilizzato è il **modello Bell-La Padula**, quindi cominceremo da questo (Bell e La Padula, 1973). Questo modello è stato progettato per gestire la sicurezza in ambiente militare, ma si può anche applicare ad altre organizzazioni. Nel mondo militare, i documenti (oggetti) possono avere un livello di sicurezza, quali non classificato, confidenziale, segreto e top secret. Anche le persone sono assegnate a questi livelli, a seconda dei documenti che è consentito loro esaminare: un generale potrebbe avere il permesso di esaminare tutti i documenti, mentre un tenente potrebbe essere limitato ai documenti confidenziali o di livelli inferiori. Un processo che viene eseguito per conto di un utente acquisisce il livello di sicurezza dell'utente e siccome vi sono livelli multipli di sicurezza, questo schema viene chiamato **sistema di sicurezza multilivello**.

Il modello Bell-La Padula pone delle regole su come le informazioni possano circolare:

- La proprietà di semplice sicurezza:** un processo in esecuzione al livello di sicurezza  $k$  può leggere solamente oggetti al suo livello o a quelli inferiori. Ad esempio, un generale può leggere i documenti del tenente, mentre un tenente non può leggere i documenti del generale.
- La proprietà \* :** un processo in esecuzione al livello di sicurezza  $k$  può scrivere solamente oggetti al suo livello o a quelli superiori. Ad esempio, un tenente può aggiungere un messaggio al mailbox del generale, per dire ciò che conosce, mentre un generale non può aggiungere un messaggio al mailbox del tenente, dicendo ciò che sa, perché potrebbe avere visto dei documenti top secret che non devono essere noti al tenente.

Riassumendo in maniera spartana, i processi possono leggere verso il basso e scrivere verso l'alto, ma non il contrario. Se il sistema applica rigorosamente queste due proprietà, si può dimostrare che nessuna informazione riuscirà a fuoriuscire da un livello più alto per andare in uno più basso. La proprietà \* è stata chiamata così, perché nell'articolo originale, gli autori non riuscivano a trovare un nome appropriato e usarono temporaneamente \* per indicarla, fino a che non avessero trovato un nome migliore; in realtà non lo trovarono mai e l'articolo fu stampato con \*. In questo modello i processi leggono e scrivono oggetti, ma non comunicano direttamente tra loro. Il modello Bell-La Padula è mostrato graficamente in Figura 9.31.

In questa figura una freccia (continua) da un oggetto ad un processo indica che il processo sta leggendo l'oggetto, ossia l'informazione si sta muovendo dall'oggetto al processo; allo stesso modo, una freccia (tratteggiata) da un processo ad un oggetto indica che il processo sta scrivendo sull'oggetto, ossia, l'informazione si sta muovendo dal processo all'oggetto. In questo modo tutte le informazioni si muovono nella direzione delle frecce; ad esempio, il processo  $B$  può leggere dall'oggetto 1, ma non dall'oggetto 3.

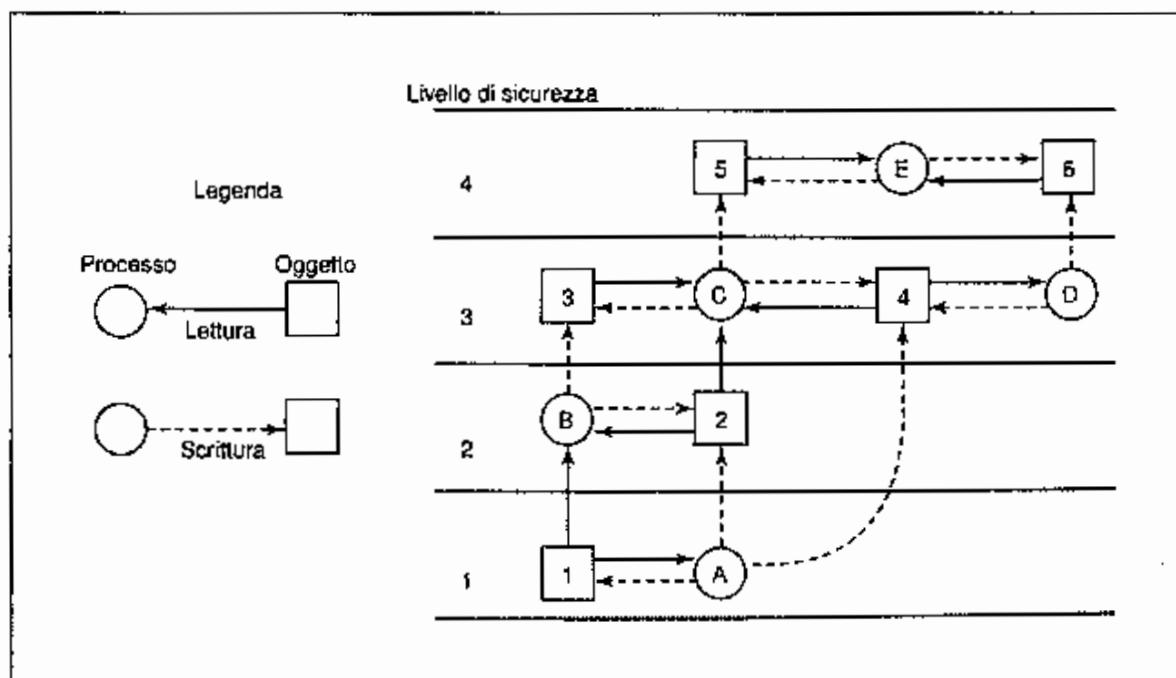


Figura 9.31 Il modello di sicurezza multilivello Bell-La Padula.

La proprietà di semplice sicurezza afferma che tutte le frecce continue (lettura) si muovono di lato o verso l'alto; la proprietà \* afferma che tutte le frecce tratteggiate (scritture) si muovono di lato o verso l'alto. Poiché l'informazione procede in orizzontale o verso l'alto, le informazioni che compaiono al livello  $k$ , non potranno mai apparire ad un livello inferiore; in altre parole, non ci sarà mai un percorso che porti le informazioni verso il basso, garantendo in questa maniera la sicurezza del modello.

## Il modello Biba

Per riassumere il modello Bell-La Padula in termini militari, un tenente può chiedere ad un privato di rivelargli tutto ciò che sa e poi copiare questa informazione nel file del generale senza violare la sicurezza. Ora mettiamo il modello in termini civili; immaginate un'azienda in cui gli uscieri abbiano livello di sicurezza 1, i programmatori abbiano livello di sicurezza 3 e il presidente della azienda abbia livello di sicurezza 5. Utilizzando Bell-La Padula, un programmatore può chiedere ad un usciere i piani futuri dell'azienda e poi sovrascrivere i file del presidente che contiene le strategie della società. Non tutte le ditte potrebbero essere ugualmente entusiaste di questo modello.

Il problema del modello Bell-La Padula è che è stato concepito per mantenere i segreti, non per garantire l'integrità dei dati; perciò per garantire l'integrità dei dati, abbiamo precisamente bisogno delle proprietà opposte (Biba, 1977):

1. **La proprietà di semplice sicurezza:** un processo in esecuzione al livello di sicurezza  $k$  può scrivere solamente oggetti al suo livello o a quelli inferiori (nessuna scrittura verso l'alto).
2. **La proprietà di integrità \***: un processo in esecuzione al livello di sicurezza  $k$  può leggere solamente oggetti al suo livello o a quelli superiori (nessuna lettura verso il basso).

Assieme, queste due proprietà assicurano che il programmatore può aggiornare i file del portiere con informazioni acquisite dal presidente, non viceversa. Naturalmente alcune organizzazioni vogliono sia le proprietà di Bell-La Padula sia quelle di Biba, ma queste sono in conflitto tra loro e sono difficilmente realizzabili simultaneamente.

## 9.7.4 L'Orange Book

Con queste premesse, non dovrebbe sorprendere che il Dipartimento della Difesa degli Stati Uniti abbia fatto molti sforzi nell'area dei sistemi sicuri; in particolare, nel 1985, pubblicò un documento, formalmente noto come Dipartimento della Difesa standard DoD 5200.28, ma solitamente chiamato **Orange Book** a causa della sua copertina, in cui si dividono i sistemi operativi in sette categorie, a seconda delle loro proprietà di sicurezza. Sebbene lo standard sia stato sostituito da un altro (molto più complicato), è ancora una guida utile per alcune proprietà di sicurezza; inoltre si può ancora vedere qualche venditore che dichiara la conformità ad alcuni livelli di sicurezza dell'Orange Book, del quale viene fornita una tabella dei requisiti nella Figura 9.32. Qui sotto invece esamineremo le categorie della sicurezza, sottolineando i punti principali.

La conformità al livello D è facile da raggiungere: non vi sono requisiti di sicurezza; comprende tutti i sistemi che hanno fallito anche il minimo test di sicurezza. MS-DOS, e Windows 95/98/Me sono al livello D.

Il livello C è destinato ad ambienti in cui gli utenti interagiscono; C1 richiede un

sistema operativo eseguito in modo protetto, login degli utenti autenticate e la possibilità per l'utente di specificare quali file possano essere disponibili agli altri utenti e come (controllo discrezionale dell'accesso). Inoltre sono richiesti un minimo di test e documentazione di sicurezza.

C2 aggiunge il requisito che il controllo discrezionale dell'accesso è al livello del singolo utente; richiede inoltre che gli oggetti (ad esempio, file, pagine di memoria virtuale) dati agli utenti, debbano essere inizializzati tutti a zero, e richiede un minimo di verifiche. Lo schema *rwx* di UNIX soddisfa C1, ma non C2. Per questo sono necessari schemi più elaborati, come ACL o equivalenti.

I livelli A e B richiedono che a tutti gli utenti e gli oggetti controllati sia assegnata un'etichetta di sicurezza, come non classificato, segreto, top secret. Il sistema deve essere in

| Criteria                                               | D | C1 | C2 | B1 | B2 | B3 | A1 |
|--------------------------------------------------------|---|----|----|----|----|----|----|
| <b>Politica di sicurezza</b>                           |   |    |    |    |    |    |    |
| Controllo dell'accesso discrezionale                   | X | X  | →  | →  | X  | →  |    |
| Riutilizzo dell'oggetto                                |   | X  | →  | →  | →  | →  |    |
| Etichette                                              |   |    | X  | X  | →  | →  | →  |
| Integrità delle etichette                              |   |    |    | X  | →  | →  | →  |
| Esportazione delle informazioni etichettate            |   |    |    | X  | →  | →  | →  |
| Etichettatura dell'output umano leggibile              |   |    |    | X  | →  | →  | →  |
| Controllo dell'accesso obbligatorio                    |   |    |    | X  | X  | →  | →  |
| Etichette di classificazione di sicurezza              |   |    |    |    | X  | →  | →  |
| Etichette dei dispositivi                              |   |    |    |    | X  | →  | →  |
| <b>Responsabilità</b>                                  |   |    |    |    |    |    |    |
| Identificazione                                        | X | X  | X  | →  | →  | →  | →  |
| Verifica                                               |   | X  | X  | X  | X  | →  | →  |
| Percorso fidato                                        |   |    |    | X  | X  | →  | →  |
| <b>Assicurazione</b>                                   |   |    |    |    |    |    |    |
| Architettura del sistema                               | X | X  | X  | X  | X  | →  | →  |
| Integrità del sistema                                  | X | →  | →  | →  | →  | →  | →  |
| Test di sicurezza                                      | X | X  | X  | X  | X  | X  | X  |
| Specifiche di progetto e di verifica                   |   |    |    | X  | X  | X  | X  |
| Analisi dei canali nascosti                            |   |    |    |    | X  | X  | X  |
| Gestione di caratteristiche fidate                     |   |    |    |    | X  | X  | →  |
| Gestione di configurazione                             |   |    |    |    | X  | →  | X  |
| Recupero fidato                                        |   |    |    |    |    | X  | →  |
| Distribuzione fidata                                   |   |    |    |    |    |    | X  |
| <b>Documentazione</b>                                  |   |    |    |    |    |    |    |
| Guida alle caratteristiche della sicurezza dell'utente | X | →  | →  | →  | →  | →  | →  |
| Manuale delle installazioni fidato                     | X | X  | X  | X  | X  | →  | →  |
| Documentazione dei test                                | X | →  | →  | X  | →  | X  |    |
| Documentazione del progetto                            | X | →  | X  | X  | X  | X  | X  |

**Figura 9.32** Criteri di sicurezza dell'Orange Book. Il simbolo X significa che vi sono nuovi requisiti. Il simbolo → significa che sono applicati i requisiti della categoria inferiore.

grado di applicare il modello di Bell-La Padula per il flusso di informazioni.

B2 aggiunge il requisito che il sistema sia stato progettato in modo modulare e stile top-down, ed il progetto deve essere presentato in modo da essere verificato; i canali nascosti (prossima sezione) devono essere analizzati.

B3 contiene tutte le funzionalità di B2, ed inoltre vi devono essere le ACL per utenti e gruppi, deve essere presente un TCB formale, così come un'adeguata verifica della sicurezza e recupero sicuro dei dati dopo un crash.

A1 richiede un modello formale di protezione del sistema ed una dimostrazione che il modello sia corretto; richiede anche una dimostrazione che l'implementazione sia conforme al modello; infine devono essere analizzati formalmente anche i canali nascosti.

### 9.7.5 Canali nascosti

Tutte queste idee di modelli formali e di sistemi che si dimostrano sicuri suonano bene, ma funzionano davvero? In una parola: no! Anche in un sistema alla cui base vi è un appropriato modello di sicurezza, che si è dimostrato essere sicuro ed è stato correttamente implementato, vi possono ancora essere delle falliche nella sicurezza. In questa sezione discutiamo come le informazioni possono trapelare anche quando è stato rigorosamente provato che ciò è matematicamente impossibile: queste idee sono dovute a Lampson (1973).

Il modello di Lampson è stato originariamente formulato in termini di un unico sistema time-sharing, ma la stessa idea può essere adattata alle LAN e altri ambienti multiutente. Nella sua forma più pura, il modello di Lampson coinvolge tre processi su alcune macchine protette: il primo processo è il cliente, che vuol far eseguire un lavoro da un secondo processo, il server; ma il cliente e il server non si fidano completamente uno dell'altro. Ad esempio, il lavoro del server è di aiutare i clienti a compilare i loro moduli delle tasse ed i clienti temono che il server memorizzi segretamente i loro dati finanziari, ad esempio, mantenendo una lista segreta di chi guadagna quanto, e poi vendendo tale lista. Al contempo, il server teme che i clienti tentino di rubare il programma che calcola le tasse.

Il terzo processo è il collaboratore, che sta cospirando davvero con il server allo scopo di rubare i dati confidenziali del cliente e, di solito, il collaboratore e il server sono posseduti dalla stessa persona. Questi tre processi sono mostrati in Figura 9.33. L'oggetto di questo esercizio è quello di progettare un sistema in cui il server non possa comunicare

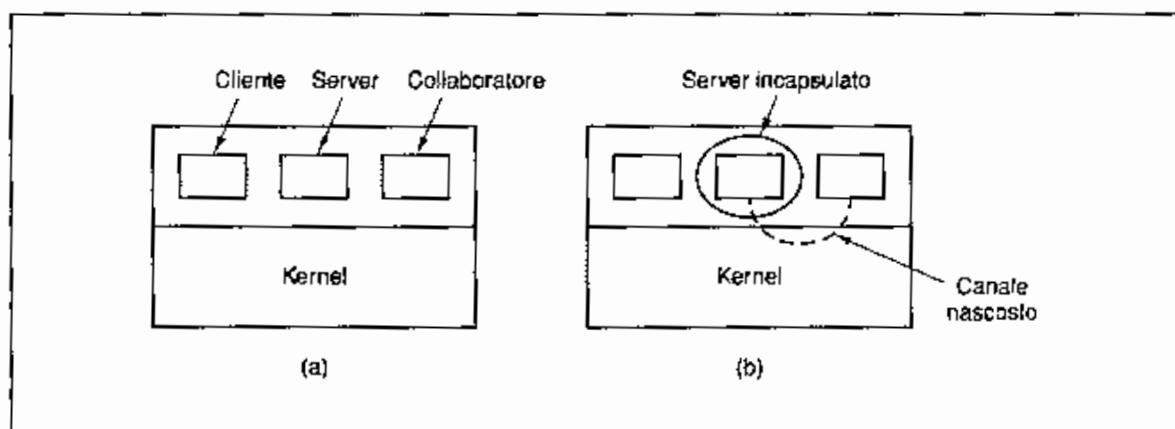


Figura 9.33 [a] I processi cliente, server e collaboratore. [b] Il server incapsulato può ancora comunicare col collaboratore tramite canali nascosti.

al collaboratore le informazioni che ha ricevuto legittimamente dal cliente. Lampson lo chiama un problema di restrizione (confinement problem).

Dal punto di vista del progettista del sistema, l'obiettivo consiste nell'incapsulare o confinare il server in modo tale che egli non possa passare l'informazione al collaboratore. Usando uno schema di matrice di protezione si può garantire facilmente che il server non possa comunicare con il collaboratore, scrivendo in un file a cui il collaboratore ha accesso di lettura; probabilmente si può anche assicurarsi che il server non possa comunicare con il collaboratore usando il meccanismo di comunicazione tra processi del sistema.

Sfortunatamente esiste la possibilità di usare canali di comunicazione più subdoli: ad esempio, il server può tentare di comunicare un flusso di bit nel seguente modo: per mandare un bit 1, calcola il più possibile per un intervallo fisso di tempo, mentre per mandare un bit 0 si mette a riposo per la stessa quantità di tempo.

Il collaboratore può provare a intercettare il flusso di bit controllando accuratamente il suo tempo di risposta: in generale, questi riceverà una risposta migliore quando il server sta trasmettendo uno 0 di quando trasmette un 1. Questo canale di comunicazione è conosciuto come **canale nascosto** ed è mostrato in Figura 9.33(b).

Naturalmente il canale nascosto è un canale rumoroso, contenente numerose informazioni spurious, ma l'informazione si può trasmettere attendibilmente su un canale di questo tipo usando un codice correttore di errore (per esempio, il codice di Hamming o anche qualcosa di più sofisticato). L'uso di un codice correttore di errore riduce ancora di più la già piccola larghezza di banda del canale nascosto, che tuttavia può essere abbastanza ampia per sottrarre informazioni significative. È abbastanza ovvio che nessun modello di protezione basato su una matrice di oggetti e di domini è sufficiente ad impedire questo tipo di fuga di informazioni.

La modulazione dell'uso della CPU non è l'unico canale nascosto; si può anche modulare il tasso di paginazione (molti fallimenti di pagina per un 1, nessuno per uno 0). In effetti, quasi tutti i modi possibili di degrado delle prestazioni del sistema in un tempo determinato sono buoni candidati: se il sistema fornisce un modo per bloccare i file, il server può bloccare qualche file per indicare un 1 e sbloccarlo per indicare uno 0. In alcuni sistemi, può essere possibile riconoscere un blocco anche su un file a cui è negato l'accesso. Questo canale coperto è mostrato in Figura 9.34, dove il file è bloccato o meno per degli intervalli di tempo fissati e noti sia al server che al collaboratore; in questo esempio, è stata trasmessa la stringa 11010100.

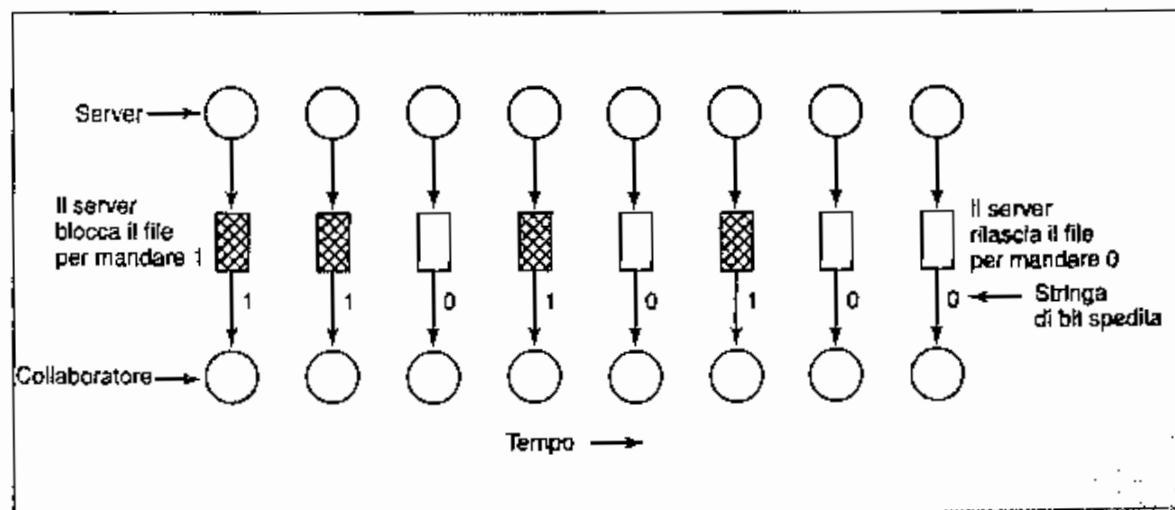


Figura 9.34 Un canale nascosto che usa il bloccaggio di file.

Anche se  $S$  non è un canale particolarmente rumoroso, bloccare e rilasciare il file concordato in precedenza richiede una temporizzazione abbastanza accurata a meno che la velocità di trasmissione dei bit non sia molto lenta; l'affidabilità e le prestazioni possono essere ancor più incrementate utilizzando un protocollo di riconoscimento. Questo protocollo utilizza due file in più,  $F1$  e  $F2$ , bloccati da server e collaboratore rispettivamente, in modo che i processi rimangano sincronizzati; dopo che il server ha bloccato o rilasciato  $S$ , modifica lo stato di lock di  $F1$  per indicare che il bit è stato spedito. Non appena il collaboratore legge il bit, modifica lo stato di lock di  $F2$  e comunica al server di essere pronto per un altro bit ed attende finché viene modificato di nuovo  $F1$ , per indicare che un altro bit è presente in  $S$ . Siccome non è più coinvolta la temporizzazione, questo protocollo è pienamente affidabile, anche in un sistema carico, e può procedere alla stessa velocità di schedulazione dei due processi. Per ottenere una larghezza di banda più ampia, è possibile utilizzare due file per unità di tempo, o rendere il canale grande un byte con otto file di segnalazione, da  $S0$  a  $S7$ .

Per la segnalazione si possono usare anche l'acquisizione ed il rilascio di risorse dedicate (periferiche a nastro, plotter, eccetera): il server acquisisce la risorsa per trasmettere un 1 e la rilascia per trasmettere uno 0. In UNIX, il server potrebbe creare un file per indicare un 1 e rimuoverlo per indicare uno 0; il collaboratore potrebbe usare la chiamata di sistema `access` per vedere se il file esiste. Questa chiamata funziona anche se il collaboratore non ha il permesso di usare il file. Sfortunatamente esistono molti altri canali nascosti.

Lampson parla anche di un modo per sottrarre informazioni al proprietario (umano) del processo server; presumibilmente il processo server sarà programmato per dire al suo proprietario quanto lavoro ha svolto a favore del cliente, così da poter emettere la fattura al cliente. Se il costo del calcolo fosse, ad esempio, 100 dollari e il reddito del cliente fosse 53000 dollari, il server potrebbe presentare al suo proprietario un conto di 100,53.

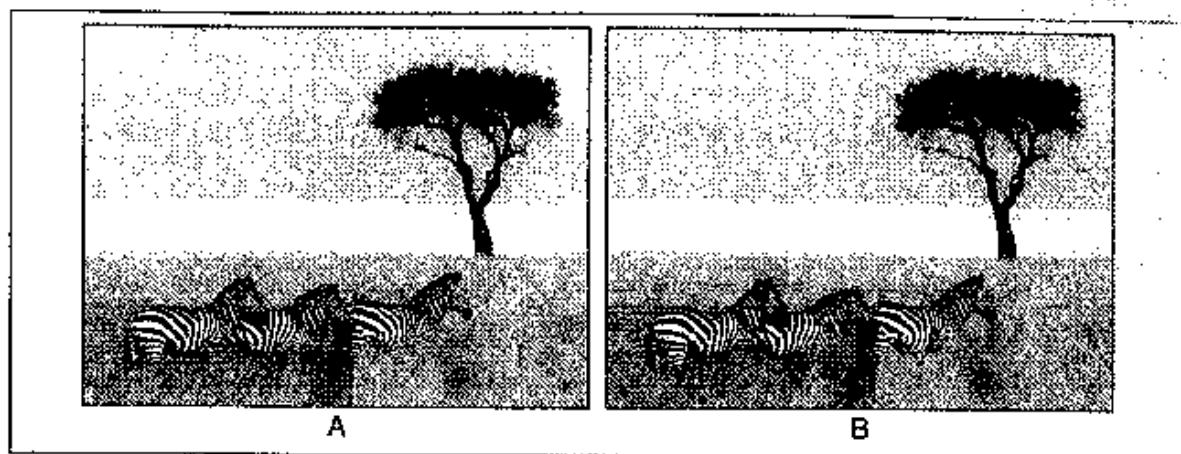
La semplice individuazione di tutti i canali nascosti, anche escludendo le azioni necessarie a neutralizzarli, è estremamente difficile. In pratica non vi è molto da fare. Introdurre un processo che provoca casualmente un errore di pagina, o altrimenti passa molto tempo di calcolo per degradare l'esecuzione del sistema e quindi ridurre la larghezza di banda di un canale nascosto non è una proposta attraente.

In precedenza abbiamo supposto che il client ed il server siano processi separati; un altro caso è quello in cui vi è un solo processo, il cliente, che sta eseguendo un programma che contiene un cavallo di Troia: il cavallo di Troia potrebbe essere stato scritto dal collaboratore col proposito di farlo eseguire dall'utente, in modo da sottrarre dei dati che il sistema di protezione impedisce siano presi direttamente dal collaboratore.

Si può usare un genere di canale nascosto lievemente differente per passare informazioni segrete tra processi, anche se un controllore umano o automatico controlla tutti i messaggi tra processi e può porre un veto a quelli sospetti: ad esempio, consideriamo un'azienda che faccia controllare manualmente tutte le e-mail in uscita, spedite dagli impiegati, per essere sicuri che non stiano diffondendo segreti a complici o concorrenti esterni all'azienda; esiste per un impiegato il modo di fare passare una considerevole quantità di informazioni confidenziali sotto il naso del controllore? Esiste!

Come esempio di questo, consideriamo la Figura 9.35(a): questa fotografia, scattata dall'autore in Kenya, contiene tre zebre che contemplano un'acacia. La Figura 9.35(b) sembra mostrare le stesse tre zebre e l'acacia, ma ha qualcosa in più: contiene, incastrato in esso, il testo completo e smontato di cinque lavori di Shakespeare, ossia Amleto, Re Lear, Macbeth, Il Mercante di Venezia, e Giulio Cesare. Tutti assieme questi lavori, superano i 700 KB di testo.

Come funziona questo canale nascosto? Il colore originale dell'immagine è 1024x768



**Figura 9.35** (a) Tre zebre ed un albero (b) Tre zebre, un albero ed il testo completo di cinque lavori di William Shakespeare.

pixel; ogni pixel è composto da tre numeri di 8 bit, uno per ogni intensità di rosso, verde e blu di ogni pixel ed il colore del pixel è dato dalla sovrapposizione lineare dei tre colori. Il metodo di codifica utilizza i bit meno significativi di ogni valore RGB del colore come canale nascosto, così ogni pixel ha posto per 3 bit di informazioni segrete, uno nel valore rosso, uno in quello verde ed uno in quello blu. Con un'immagine di questa dimensione, vi si possono immagazzinare fino a 1024x768x3 bit ossia 294912 byte di informazioni segrete.

Il testo completo dei cinque lavori ed una breve nota raggiungono i 734891 byte; dapprima si è proceduto a comprimere questi dati con un algoritmo di compressione standard, giungendo a circa 274 Kb. L'output così compresso è stato poi crittato ed inserito nei bit meno significativi di ogni valore di colore. Come si può vedere (o meglio, come non si può vedere), l'esistenza di tale informazione è completamente invisibile: l'occhio non può distinguere colori a 7 bit da colori ad 8 bit. Una volta che l'immagine ha passato il censore, il ricevente elimina i bit meno significativi, applica la decriptazione, l'algoritmo di decompressione e recupera i 734891 byte originali. Il nascondere l'esistenza di informazione in questo modo è detto **steganografia** (dal greco "scrittura nascosta"). La steganografia non è popolare tra i governi che cercano di limitare le comunicazioni tra i loro cittadini, ma è molto popolare tra le persone che credono fortemente nella libera comunicazione.

Il vedere le due immagini in bianco e nero ed a bassa risoluzione non rende giustizia della potenza di tale tecnica; per avere un'idea migliore di come funzioni la steganografia, l'autore ha preparato una dimostrazione, che contiene l'immagine a pieni colori di Figura 9.35(b), con i cinque lavori incastrati in essa. La dimostrazione si può trovare presso [www.cs.vu.nl/~astl/](http://www.cs.vu.nl/~astl/), seguendo il link covered\_writing, alla voce STEGANOGRAPHY DEMO. Seguite quindi le istruzioni presenti nella pagina per scaricare l'immagine ed il software di steganografia necessario per estrarre i lavori.

Un altro uso della steganografia consiste nell'inserire dei segni di riconoscimento nascosti (watermark, letteralmente marchi sull'acqua) nelle immagini utilizzate in pagine Web, in modo da poter rilevare il loro furto e riutilizzo in altre pagine Web; se la vostra pagina Web contiene un'immagine con messaggio segreto, ad esempio, Copyright 2000 - General Images Corporation, sarebbe difficile convincere un giudice che voi abbiate prodotto da soli tale immagine! Anche la musica, i film e altri generi di materiali possono essere segnati in questo modo.

Naturalmente il fatto che siano utilizzati i watermark, incoraggia alcune persone a cercare il modo di rimuoverli: uno schema che immagazzini informazioni nei bit meno significativi di ogni pixel può essere sconfitto ruotando l'immagine di 1 grado in senso orario, convertendola poi in un formato meno preciso, come JPEG, ruotandola nuovamente di un grado in senso antiorario, riconvertendola infine nel suo formato originale (ad esempio, gif, bmp, tif). La conversione in JPEG metterà in disordine i bit meno significativi, mentre le rotazioni causeranno un massiccio uso di calcoli in virgola mobile, che inseriranno errori di arrotondamento e più in generale aggiungeranno rumore sui bit meno significativi. Le persone che utilizzano questi segni di riconoscimento lo sanno (o dovrebbero saperlo), perciò inseriscono informazioni ridondanti all'interno dei loro copyright ed utilizzano altri schemi oltre l'uso dei bit meno significativi dei pixel. A questo punto, chi attacca è stimolato a cercare nuove tecniche di rimozione e così via.

## 9.8 Ricerche sulla sicurezza

La sicurezza dei sistemi operativi è un argomento molto interessante, ma nonostante le numerose ricerche in corso, la maggior parte non è direttamente correlata ai sistemi operativi; al contrario, tali ricerche riguardano la sicurezza delle reti (ad esempio, e-mail, Web, sicurezza nell'e-commerce), crittografia, Java o la gestione di un'installazione di computer sicura.

Tuttavia esiste un'altra parte della ricerca che è più vicina al nostro argomento; ad esempio, è ancora importante l'autenticazione dell'utente. Monroe e Rubin (1997) l'hanno studiata tramite la dinamica dell'uso della tastiera, Pentland e Choudhury (2000) hanno discusso il riconoscimento delle facce, e Mark (2000) ha sviluppato un modo per modellarlo, oltre ad altri.

Altri lavori dedicati alla sicurezza dei sistemi operativi sono i seguenti: Bershad et al. (1995a) hanno sostenuto che la protezione è un problema software e non hardware (ad esempio, della MMU); Mazieres et al. (1999) hanno cercato i file system distribuiti sicuri; Myers e Liskov (1997) hanno studiato i modelli dei flussi di informazioni sicuri; Chase et al. (1994) hanno esaminato la sicurezza in sistemi con un vasto spazio di indirizzamento occupato da processi multipli; la sicurezza con le smart card è stata trattata da Clark e Hoffman (1994), mentre Goldberg et al. (1998) hanno costruito la filogenia dei virus.

## 9.9 Sommario

Si possono minacciare i sistemi operativi in diversi modi, spaziando da attacchi dall'interno a virus che provengono dall'esterno. Molti attacchi iniziano quando un cracker cerca di penetrare in uno specifico sistema, spesso indovinando le password: questi attacchi spesso utilizzano dizionari di password comuni ed hanno uno straordinario successo. Ma la sicurezza della password può essere aumentata utilizzando salt, password usa e getta, e schemi challenge-response; altre possibilità sono date da smart card, indicatori biometrici e analizzatori retinici.

Sono noti differenti attacchi che si possono portare ai sistemi operativi: cavalli di Troia, login spoofing, bombe logiche, porte nascoste, overflow dei buffer. Gli attacchi generici comprendono: richiedere memoria e curiosare in essa, effettuare delle chiamate di siste-

ma illegali per vedere cosa succede, o anche cercare di raggiungere chi è all'interno, per ottenere informazioni che non dovrebbero rivelare.

I virus stanno diventando un problema sempre più serio per molti utenti e si presentano in varie forme: virus residenti in memoria, virus del settore d'avvio, virus delle macro e così via; utilizzare un analizzatore di virus per riconoscere le firme dei virus è utile, ma i virus davvero validi possono crittare gran parte del loro codice e modificare la parte restante ad ogni loro copia, rendendo veramente difficile rilevarli. Alcuni software antivirus agiscono non solo cercando le firme dei virus, ma andando alla ricerca di certi comportamenti sospetti. Evitare i virus, in primo luogo grazie a pratiche informatiche sicure, è meglio che cercare di affrontare l'attacco; in breve, non caricate ed eseguite programmi la cui origine è sconosciuta e la cui affidabilità è discutibile.

Un'altra questione che si deve trattare in questi giorni è il codice mobile ed i possibili approcci sono: metterlo in sandbox, interpretarlo, o eseguire solo quel codice che proviene da fonti sicure.

Si possono proteggere i sistemi utilizzando una matrice di protezione, con i domini (ad esempio, utenti) in verticale e gli oggetti in orizzontale; la matrice può essere trattata per righe, portando a sistemi basati su capability, o per colonne, portando a sistemi basati su ACL.

Si possono progettare sistemi sicuri, ma deve essere un obiettivo sin dall'inizio; probabilmente la regola di progettazione più importante è di avere un TCB che non si può aggirare, qualsiasi risorsa si acceda. La sicurezza multilivello si può basare sul modello Bell-La Padula, progettato per mantenere i segreti, o sul modello Biba, progettato per mantenere l'integrità del sistema. L'Orange Book descrive i requisiti che deve avere un sistema fidato; infine, anche se un sistema si è dimostrato essere sicuro, si deve fare attenzione ai canali nascosti, che possono facilmente sovvertire il sistema, creando dei canali di comunicazione non inclusi nel modello.

## PROBLEMI

- Consideriamo un cifrario a chiave segreta che abbia una matrice 26x26, le cui righe e colonne siano indicate da ABC..Z; il messaggio in chiaro viene crittato due lettere alla volta, il primo carattere è la colonna, il secondo la riga; la cella formata dall'intersezione di riga e colonna contiene due caratteri cifrati. Quali vincoli si devono porre alla matrice e quante chiavi vi sono?
- Risolvere il seguente cifrario monoalfabetico; il messaggio in chiaro, composto da sole lettere, è un brano conosciuto tratto da una poesia di Lewis Carroll:

```
kfd ktbd fzm eubd kfd pzyiom mzttx ku kzyg ur bzha kfthcm
ur mfudm zhx mftmn zhx mdzythc pzq ur ezsscdm zhx gthcm
ztx pfa kfd mdz tm sutythc fuk zhx pfdkdi ntcn fzld pthcm
sok pztk z stk kfd uamkdim eitdx sruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
hu foia mzttx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzh
```

- Consideriamo il seguente modo per crittare un file: l'algoritmo di crittazione utilizza due array di  $n$  byte,  $A$  e  $B$ . I primi  $n$  byte sono letti dal file in  $A$ ; quindi  $A[0]$  viene copiato in  $B[j]$ ,  $A[1]$  in  $B[j+1]$ ,  $A[2]$  in  $B[k]$  eccetera. Dopo che tutti gli  $n$  byte sono stati copiati nell'array  $B$ , tale array viene scritto nel file di output e sono letti altri  $n$  byte in  $A$ . Questa procedura continua finché tutto il file viene crittato;

notiamo che la crittazione non è effettuata cambiando un carattere con un altro, ma modificandone l'ordine. Quante chiavi bisogna provare per cercare la chiave per esaustione? È vantaggioso rispetto alla sostituzione monoalfabetica?

4. La crittografia a chiave segreta è più efficiente di quella a chiave pubblica, ma richiede che il mittente e il destinatario concordino una chiave in precedenza. Supponiamo che il mittente ed il destinatario non si siano mai incontrati, ma esista una terza persona fidata che condivide una chiave segreta con il mittente ed un'altra chiave segreta col destinatario. Come fanno il mittente ed il destinatario a stabilire una nuova chiave segreta tra loro in queste circostanze?
5. Dare un esempio semplice di funzione matematica che in prima approssimazione si comporta come una funzione unidirezionale.
6. Non avere la visualizzazione della password sullo schermo è più sicuro che avere degli asterischi per ogni carattere digitato, poiché il secondo metodo mostra la lunghezza della password a chiunque sia nelle vicinanze dello schermo. Supponendo che la password sia formata solamente da lettere maiuscole, minuscole e numeri, e che sia almeno lunga 5 caratteri, ed al massimo lunga 8, quanto è più sicuro non visualizzare nulla?
7. Dopo esserti laureato, trovi lavoro come direttore in un grande centro di calcolo dell'Università che ha appena messo da parte il suo vecchio sistema operativo su mainframe in cambio di un grande server LAN che esegue UNIX. Quindici minuti dopo l'inizio del lavoro, il tuo assistente irrompe nel tuo ufficio urlando: "Alcuni studenti hanno scoperto l'algoritmo che usiamo per crittografare le password e l'hanno messo in Internet!" Cosa dovrà fare?
8. Lo schema di protezione di Morris-Thompson con numeri casuali di  $n$  bit (salt) è stato progettato per rendere difficile a un intruso scoprire un gran numero di password crittografando anticipatamente stringhe comuni. Lo schema offre protezione anche da un utente (studente) che prova a indovinare la password del super-user sulla sua macchina? Supporre che il file delle password sia leggibile.
9. Elencare tre caratteristiche che un buon indicatore biometrico deve avere per essere utile per autenticare il login.
10. Esiste un modo ragionevole per usare l'hardware MMU ed evitare attacchi con overflow del tipo mostrato in Figura 9.11? Spiegare perché o perché no.
11. Un Dipartimento di Informatica ha un gran numero di macchine UNIX sulla sua rete locale. Gli utenti su ogni macchina possono usare un comando della forma  
`machine4 who`  
 che può essere eseguito sulla `machine4` senza che sia necessario che l'utente acceda alla macchina remota. Ciò è implementato facendo sì che il kernel dell'utente invii il comando e il suo TELNET alla macchina periferica. Questo schema è sicuro se i kernel sono tutti fidati? E se alcune di queste macchine sono personal computer di studenti senza protezione hardware?
12. Quale proprietà hanno in comune l'implementazione di password in UNIX e lo schema di Lamport per eseguire login su una rete non sicura?
13. Lo schema delle password usa e getta di Lamport utilizza le password in ordine inverso. Non sarebbe più semplice utilizzare  $f(s)$  la prima volta,  $f(f(s))$  la seconda volta e così via?

14. Visto che gli Internet Cafe diventano sempre più diffusi, la gente avrà sempre più desiderio di andare ovunque e fare affari da lì. Descrivere un modo per generare documenti firmati da chi sta usando una smart card (supponendo che tutti i computer siano dotati di un lettore di smart card). È uno schema sicuro?
15. Un attacco con cavallo di Troia può riuscire in un sistema protetto da capability?
16. Elencare una caratteristica del compilatore C che potrebbe eliminare un gran numero di fallo nella sicurezza. Perché non è implementata molto più spesso?
17. Quando si rimuove un file, i suoi blocchi sono generalmente risistemati nella lista libera, ma non sono cancellati. Pensi che sarebbe una buona idea far sì che il sistema operativo cancelli ogni blocco prima di rilasciarlo? Nella tua risposta considera i fattori di sicurezza e di efficienza, spiegando l'effetto di ognuno.
18. Come si può modificare TENEX per non avere il problema della password descritto nel testo?
19. Come può un virus parassita (a) essere sicuro che sarà eseguito prima del programma ospite e (b) che lascerà il controllo a tale programma dopo avere fatto quello che fa?
20. Alcuni sistemi operativi richiedono che le partizioni dei dischi debbano cominciare all'inizio di una traccia. Come questo fatto facilita la vita dei virus del settore d'avvio?
21. Modificare il programma di Figura 9.13 in modo che trovi tutti i programmi C, invece di tutti i file eseguibili.
22. Il virus di Figura 9.16(d) è crittato. Come fanno gli scienziati appositi dei laboratori antivirus a dire quale parte del file sia la chiave, per poi decrittarlo e ottenerne il sorgente? Come fa Virgil a rendere il loro lavoro un po' più difficile?
23. Il virus di Figura 9.16(c) possiede sia un compressore che un decompressore. Il decompressore è necessario per espandere ed eseguire il programma eseguibile compresso. A cosa serve il compressore?
24. Elencare uno svantaggio del crittare un virus polimorfo *dal punto di vista del creatore del virus*.
25. Spesso si danno le seguenti istruzioni per ricuperare l'attacco di un virus:
  1. Avviare il sistema infetto.
  2. Salvare i file su un supporto esterno.
  3. Eseguire fdisk per formattare il disco.
  4. Reinstallare il sistema operativo con il CD-ROM originale.
  5. Ricaricare i file dal supporto esterno.Dire quali sono i due gravi errori in queste istruzioni.
26. Esistono virus compagni (virus che non modificano i file esistenti) in UNIX? Se è così, come? Se no, perché no?
27. Qual è la differenza tra un virus e un worm? Come si riproducono?
28. Gli archivi auto-estraenti, che contengono uno o più pacchetti di file compressi con un programma di estrazione, sono usati di frequente per spedire programmi o aggiornamenti di programmi. Discutere le implicazioni per la sicurezza dovute all'uso di questa tecnica.

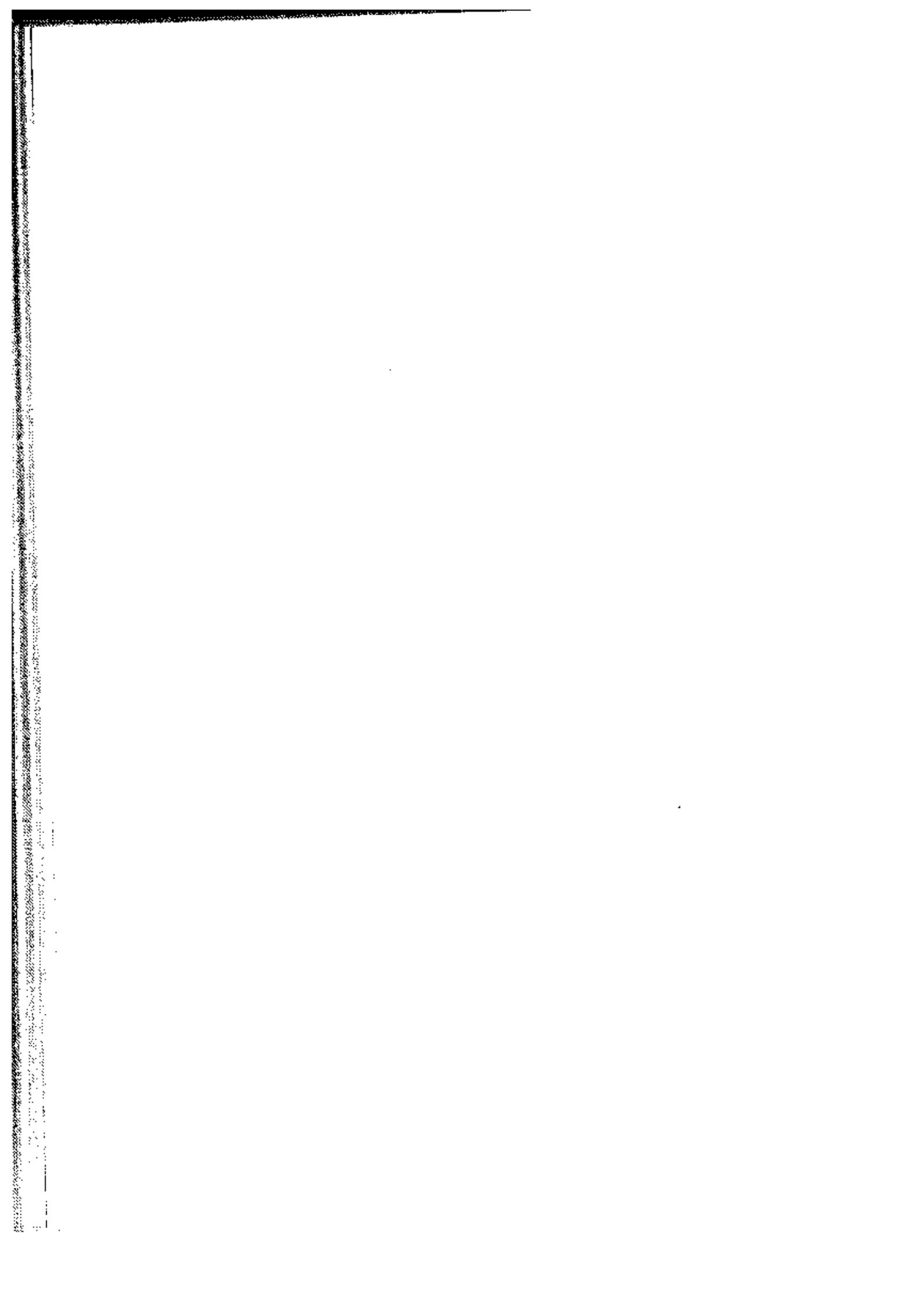
29. Su qualche macchina l'istruzione SHR, utilizzata in Figura 9.18 (b), riempie i bit non utilizzati con zero; su altre il segno del bit è esteso a destra. Per la correttezza della Figura 9.18(b), è importante il tipo di istruzione di shift a destra utilizzato? Se è così, quale è il migliore?

30. Rappresentare la proprietà ed i permessi mostrati nel listato di questa directory UNIX sotto forma di matrice di protezione. Nota: *asw* è un membro di due gruppi, *users* e *devel*; *gwm* è un membro solo di *users*. Trattare ognuno dei due utenti e dei due gruppi come domini, così la matrice avrà quattro righe (una per dominio) e quattro colonne (una per file).

```
-rw-r-r- 2 gwm users 908 May 26 16:45 PPP-Notes
-rwxr-xr-x 1 asw devel 432 May 13 12:35 prog1
-rw-rw-- 1 asw users 50094 May 30 17:51 project.t
-rw-r-- 1 asw devel 13124 May 31 14:30 splash.gif
```

31. Esprimere i permessi del problema precedente sotto forma di ACL.
32. Modificare l'ACL di un file per garantire o negare un accesso che non può essere espresso tramite il sistema UNIX *rwx*. Spiegare la modifica.
33. Per verificare che un'applet è stata firmata da un venditore fidato, il venditore può accludere un certificato firmato da una terza persona fidata che contiene la sua chiave pubblica. Tuttavia per leggere il certificato, l'intente necessita della chiave pubblica di tale persona. Una quarta persona fidata potrebbe fornirla, ma ora l'utente necessita della chiave pubblica di tale quarta persona. Sembra che non vi sia modo di avviare il sistema di verifica, però i browser esistenti lo usano: come può accadere?
34. In una matrice di controllo degli accessi piena, le righe sono i domini e le colonne sono gli oggetti: cosa succede se qualche oggetto è richiesto da due domini?
35. Si sono discussi due differenti meccanismi di protezione, cioè le capability e la lista di controllo degli accessi. Per ognuno dei seguenti problemi di protezione dire quali di questi meccanismi si può usare.
- Chiara vuole che i suoi file siano leggibili da tutti tranne che dal suo collega d'ufficio.
  - Mario e Sandro vogliono condividere dei file segreti.
  - Linda vuole che qualcuno dei suoi file sia pubblico.
36. Nella schema di Amoeba per la protezione delle capability, un utente può chiedere al server di generare una nuova capability con minori diritti, in modo che possa essere data ad un amico. Cosa succede se l'amico chiede al server di rimuovere altri diritti, in modo da poterla dare a qualcun altro?
37. In Figura 9.31 non c'è nessuna freccia dal processo B all'oggetto 1. Sarebbe permessa una tale freccia? Se no, quale regola violerebbe?
38. In Figura 9.31 non c'è nessuna freccia dall'oggetto 2 al processo A. Sarebbe permessa una tale freccia? Se no, quale regola violerebbe?
39. Se fosse consentito di mandare messaggi da un processo ad un altro in Figura 9.31, quale regola si violerebbe? In particolare il processo B a quali processi potrebbe mandare messaggi, e a quali no?
40. Considerare il sistema steganografico di Figura 9.35. Ogni pixel può essere rappresentato in uno spazio di colori da un punto in un sistema tridimensionale, con i colori R, G e B sugli assi. Usando questo spazio, spiegare cosa succede alla risolu-

- zione del colore quando viene impiegata la steganografia come in questa figura.
41. Il testo ASCII in linguaggio naturale può essere compresso del 50% almeno, utilizzando vari algoritmi di compressione. Utilizzando questa conoscenza, qual è la capacità di trasporto steganografico per un testo ASCII (in byte) di un'immagine 1600x1200, memorizzata utilizzando i bit meno significativi di ogni pixel? Quanto è incrementata la dimensione dell'immagine, se si usa questa tecnica (supponendo che non vi sia crittazione oppure nessuna espansione a causa della crittazione)? Qual è l'efficienza di questo schema, cioè il payload / (byte trasmessi)?
  42. Supponiamo che uno sparuto gruppo di dissidenti politici, che vive in un paese repressivo, stia utilizzando la steganografia per mandare messaggi al mondo esterno e descrivere in tale modo le condizioni di vita del loro paese. Il governo è a conoscenza di ciò e li sta combattendo, mandando all'esterno finte immagini contenenti falsi messaggi stenografici. Come fanno i dissidenti ad aiutare la gente a capire quali siano i veri messaggi e quali quelli falsi?
  43. Scrivere un paio di script shell per spedire e ricevere un messaggio di testo tramite un canale nascosto in un sistema UNIX. (Suggerimento: utilizzare il tempo di esecuzione dei processi come canale nascosto. Il comando *sleep* garantisce un ritardo minimo, stabilito dal suo argomento; il comando *ps* può essere utilizzato per vedere tutti i processi in esecuzione).
  44. Scrivere un paio di programmi, in C o script shell, per spedire e ricevere un messaggio tramite un canale nascosto in un sistema UNIX. (Suggerimento: Un bit di permesso può essere visto anche quando un file non è accessibile in altro modo; il comando *sleep* o una chiamata di sistema garantisce un ritardo per un tempo fissato, stabilito dal suo argomento. Misurare la velocità dei dati su un sistema inattivo; quindi creare un pesante carico, avviando numerosi processi in background e misurare di nuovo la velocità dei dati).



# CASO DI STUDIO 1: UNIX E LINUX

Nel capitolo precedente si sono esaminati molti concetti fondamentali sui sistemi operativi, astrazioni, algoritmi e tecniche varie. Adesso è il momento di guardare a come questi principi sono stati applicati nel mondo reale; incominceremo da UNIX perché gira su una varietà di computer maggiore rispetto ad ogni altro sistema operativo. Esso è il sistema operativo dominante (quasi l'unico) sulle workstation di fascia alta e sui server, ma è usato anche su altri tipi di macchine, a partire dai notebook fino ai supercomputer. È stato progettato con cura, avendo chiari in mente gli obiettivi da raggiungere, per cui, malgrado siano passati molti anni da quando fu ideato, è tuttora elegante e moderno. UNIX illustra molti principi importanti di progettazione, pochi di questi sono stati copiati dagli altri sistemi.

La presentazione di UNIX incomincerà con la storia e l'evoluzione del sistema. Successivamente se ne darà una visione generale per capire come viene usato; tale visione risulterà particolarmente interessante per i lettori familiari con Windows, visto che quest'ultimo virtualmente nasconde tutti i dettagli del sistema ai suoi utenti. Sebbene le interfacce grafiche possano risultare più semplici per gli utenti che si avvicinano per la prima volta ad un sistema operativo, esse forniscono poca flessibilità e poca visibilità su come il sistema lavora effettivamente.

Quindi arriveremo alla parte centrale di questo capitolo: l'esame dei processi, della gestione della memoria, dell'I/O, del file system, e della sicurezza in UNIX. Per ognuno di questi argomenti discuteremo prima dei concetti fondamentali, quindi delle chiamate di sistema ed infine dell'implementazione.

Un problema che incontreremo è l'esistenza di molte versioni e cloni di UNIX, compreso AIX, BSD, IBSD, HP-UX, Linux, MINIX, OSF/1, SCO UNIX, System V, Solaris, XENIX, e molti altri; ed ognuno di questi ha sviluppato molte release diverse. Fortuna-

tamente, i concetti fondamentali e le chiamate di sistema sono rimaste praticamente identiche in ogni progetto; in più, le strategie generali di implementazione, gli algoritmi, le strutture dati sono simili, ma vi sono alcune differenze. In questo capitolo vedremo parecchi esempi in fase di trattazione dell'implementazione di un certo argomento, in primo luogo 4.4BSD (che è la base per FreeBSD), System V Release 4 e Linux. Informazioni aggiuntive circa le varie implementazioni possono essere trovate in Beck et al. 1998; Goodheart e Cox, 1994; Maxwell, 1999; McKusick e al., 1996; Pate, 1996; Vahalia, 1996.

## 10.1 La storia di UNIX

UNIX ha una storia lunga ed interessante che sarà il punto di partenza del nostro studio. È interessante capire come un progetto di secondaria importanza di un giovane ricercatore è diventato presto un affare di diverse migliaia di dollari coinvolgendo aziende multinazionali, università, agenzie governative e comitati internazionali di standardizzazione. Qui di seguito si parlerà di come si è sviluppata la storia.

### 10.1.1 UNICS

Tra il 1940 e il 1950 tutti i computer erano personal computer, nel senso che il modo normale di usare un calcolatore consisteva nel prenotarlo per un'ora e lavorare con l'intera macchina per il tempo prenotato. Certamente queste macchine erano fisicamente immense, ma solamente una persona (il programmatore) poteva utilizzarle, a quei tempi. Quando furono progettati i sistemi batch, negli anni '60, il programmatore scriveva il programma su schede perforate e le portava in sala macchina, quando si erano raccolti abbastanza job, l'operatore le leggeva tutte come un unico job batch. Di solito, per ottenere il risultato era necessario attendere un'ora dal momento in cui si era consegnato il job all'operatore. In queste circostanze, i tempi di debugging erano enormi, perché una singola virgola posizionata male per errore faceva perdere parecchie ore di lavoro ai programmatori.

Il timesharing (sistema a suddivisione di tempo) fu inventato presso il Dartmouth College e l'M.I.T., per aggirare ciò che quasi tutti allora consideravano una situazione improduttiva e poco soddisfacente. Il sistema di Dartmouth girava solo con il BASIC ed ebbe un breve successo commerciale, ma il sistema del M.I.T., il CTSS, per applicazioni generali, ebbe un grosso successo tra la comunità scientifica. In poco tempo i ricercatori del M.I.T. unirono le loro forze con i laboratori della Bell e della General Electric (a quel tempo venditori di computer) e cominciarono a progettare un secondo sistema, **MULTICS** (**M**ULTiplexed **I**nformation and **C**omputing **S**ervice), come abbiamo detto nel Capitolo 1.

Sebbene la Bell fosse uno dei fondatori di MULTICS, ad un certo punto si tirò indietro, la qual cosa lasciò uno dei ricercatori della Bell, Ken Thompson, a guardarsi intorno cercando qualcosa di interessante da fare. Questi infine decise di riscrivere autonomamente una versione semplice di MULTICS (questa volta in assembler) su un minicomputer PDP-7 inutilizzato. A dispetto delle dimensioni ridotte del PDP-7, il nuovo sistema di Thompson funzionò e poté essere di supporto agli sforzi di Thompson. Di conseguenza, un altro ricercatore dei Bell Labs, Brian Kernighan, un po' per gioco chiamò quel sistema **UNICS** (**U**niplexed **I**nformation and **C**omputing **S**ervice). Nonostante un gioco di parole legato a "EUNUCHS" (eunuco), che faceva pensare a MULTICS privato di qualcosa, il nome diventò definitivo anche se in un secondo momento fu leggermente cambiato in **UNIX**.

## 10.1.2 UNIX del PDP-11

Il lavoro di Thompson impressionò a tal punto i suoi colleghi ai laboratori della Bell che egli fu subito contattato da Dennis Ritchie e più tardi da tutto il dipartimento, cosa che fece subito fare due passi avanti al suo sviluppo. Innanzitutto, UNIX fu tolto dall'inutile PDP-7 e portato su un computer più moderno, il PDP-11/20, e più tardi il PDP-11/45 e sul PDP-11/70. Queste ultime due macchine dominarono il mondo dei minicomputer per gran parte degli anni '70. Entrambi il PDP-11/45 e il PDP-11/70 erano macchine potenti con ampia memoria fisica per il loro periodo (256 KB e 2 MB, rispettivamente). Erano anche dotati di una protezione hardware della memoria che rendeva possibile servire più utenti allo stesso tempo. Ad ogni modo erano entrambe macchine a 16 bit che limitavano i processi individuali a 64 KB per spazio di istruzione e 64 KB per spazio dati, anche se la macchina poteva avere molta più memoria fisica a disposizione.

Il secondo sviluppo riguardò il linguaggio in cui era scritto UNIX. Era ovvio che dover riscrivere l'intero sistema per le nuove macchine non sarebbe stato propriamente divertente, così Thompson decise di riscrivere UNIX in un linguaggio ad alto livello da lui progettato che aveva chiamato B. B era una versione semplificata di BCPL (il quale era esso stesso una semplificazione di CPL, che, come il PL/I, non funzionò mai bene). Questo tentativo non ebbe successo, soprattutto per la mancanza di strutture del linguaggio, allora Ritchie progettò un successore di B, chiamato C, per il quale scrisse un compilatore di ottima qualità. Thompson e Ritchie riscrissero insieme UNIX nel linguaggio C. C è stato il linguaggio giusto giunto nel momento giusto, e da allora ha dominato la programmazione di sistemi.

Nel 1974, Ritchie e Thompson pubblicarono un articolo su UNIX che fu una pietra miliare (Ritchie e Thompson, 1974). Per il lavoro descritto in questo articolo più tardi essi hanno ricevuto il prestigioso Premio Turing dall'ACM (Ritchie, 1984; Thompson, 1984). La pubblicazione di quest'articolo spinse molte università a chiedere una copia di UNIX ai Bell Labs. Poiché l'azienda associata ai Bell Labs, AT&T, era in quel momento un monopolio regolato, e non poteva trattare affari nel campo dei computer, non ebbe alcun problema a cedere UNIX all'università per un prezzo modesto.

In una di quelle coincidenze che spesso fanno la storia, il PDP-11 era il computer scelto presso quasi tutti i dipartimenti di informatica delle università, ed i sistemi operativi di cui erano forniti erano considerati, sia dai professori che dagli studenti, fatti male. UNIX riempì velocemente il vuoto, anche perché fu fornito con tutto il codice sorgente, così la gente avrebbe potuto pasticciarci, cosa che fece, quanto gli pareva. Si organizzarono incontri scientifici in cui si parlava di UNIX, con diversi oratori famosi che parlavano di qualche strano difetto del kernel che avevano trovato e a cui avevano posto rimedio. Un professore australiano, John Lions, scrisse dei commenti sul codice sorgente di UNIX del genere che, di solito, si usa per i lavori di Chaucer o Shakespeare (ristampato come Lions, 1996). Il libro descriveva la versione 6, così chiamata perché essa era descritta nella sesta edizione del Manuale del Programmatore UNIX. Il codice sorgente era di 8200 righe di C e 900 righe di Assembler. Il risultato che provocò tutta questa attività fu di portare nuove idee e miglioramenti al sistema in modo rapido.

Nel giro di pochi anni la Versione 6 fu sostituita dalla Versione 7, la prima versione portabile di UNIX (essa girava sul PDP-11 e sull'Interdata 8/32), ma era costituita da 18800 righe di codice C e 2100 righe di assembler. Un'intera generazione di studenti venne allevata con la Versione 7, il che contribuì al dilagare del sistema anche nell'industria, via via che si laureavano. Dalla metà degli anni 80, UNIX era utilizzato nei mini computer e nelle workstation di parecchi produttori, perché un notevole numero di aziende acquistò la licenza del codice sorgente per contribuire a sviluppare il proprio sistema UNIX. Una di queste fu una piccola società agli inizi chiamata Microsoft, che vendette la

versione 7 sotto il nome di XENIX per un certo numero di anni, fino a quando il suo interesse non si spostò altrove.

### 10.1.3 UNIX portabile

Una volta scritto in C, far girare UNIX su macchine nuove, cioè portarlo, era molto più facile che nel passato: il trasporto richiede la scrittura di un compilatore C per la nuova macchina, e la scrittura dei driver dei dispositivi di I/O, tipo i terminali, le stampanti e i dischi; infatti, sebbene il codice sia in C non si può trasferire da una macchina a un'altra semplicemente ricompilandolo, perché non esistono due dischi che funzionino nello stesso modo. Infine si deve scrivere, in linguaggio assembler, una limitata quantità di codice dipendente dalla macchina, come il driver delle interruzioni e i programmi di gestione della memoria.

Il minicomputer Interdata 8/32 fu il primo dopo il PDP-11 su cui fu portato UNIX, ed in tale occasione vennero alla luce un gran numero di ipotesi fatte implicitamente da UNIX riguardo alle macchine su cui girava, come il fatto che gli interi erano formati da 16 bit e così anche i puntatori (il che implica una dimensione massima di un programma di 64K) e che la macchina non aveva più di tre registri per memorizzare le variabili importanti. Nessuna di queste ipotesi si rivelò buona sui calcolatori dell'Interdata, per cui fu necessario ripulire UNIX spendendo una quantità considerevole di ore di lavoro.

Un altro problema fu che sebbene il compilatore di Ritchie fosse veloce e producesse un buon codice oggetto, questo era quello del PDP-11. Piuttosto che scrivere un nuovo compilatore specifico per l'Interdata, Steve Johnson dei Bell Labs progettò e realizzò il compilatore C portabile che, con uno sforzo moderato, si poteva adeguare per produrre codice di ogni tipo di macchina. Per anni i compilatori C si sono basati su quello di Johnson, il che ha aiutato decisamente UNIX ad essere portato su nuovi calcolatori.

Il trasporto di UNIX sull'Interdata sulle prime andò lentamente perché tutto lo sviluppo delle modifiche dovette essere fatto sull'unica macchina che lavorava con UNIX, cioè il PDP-11, che era situato al quinto piano dei Bell Labs, mentre l'Interdata era al primo piano. La produzione di una nuova versione comportava il compilarsela al quinto piano e poi portarla con un nastro magnetico al primo piano per vedere se funzionava. Dopo diversi mesi si manifestò un gran interesse a connettere elettronicamente più macchine insieme: in questo periodo la rete UNIX mise le sue radici; infatti, dopo essere stato installato sull'Interdata, UNIX fu installato sui VAX e su altri computer.

Dopo che nel 1984 il governo degli Stati Uniti permise la suddivisione della AT&T, l'azienda fu legalmente libera di costituire una consociata che trattava computer, così in breve tempo mise sul mercato il suo primo prodotto commerciale di UNIX, il System III. Questo prodotto non ebbe un gran successo, per cui, un anno dopo, fu rimpiazzato da un'altra versione, il System V (cosa accadde della versione IV è uno dei misteri tuttora esistenti). Il System V originale è stato poi rimpiazzato dal System V, versione 2, 3, 4, ognuna più grande e complicata della precedente; in questo procedimento svanì l'idea originale di UNIX che era quella di avere un sistema semplice ed elegante. Sebbene il gruppo di Ritchie e Thompson più tardi produsse l'ottava, la nona e la decima edizione di UNIX, queste non furono molto usate, poiché la AT&T aveva posto tutti i suoi sforzi commerciali in questa edizione; comunque, alcune delle idee presenti nelle edizioni 8, 9 e 10 furono incorporate in System V. Infine, AT&T decise che era opportuno trasformarsi in una compagnia telefonica, non una compagnia informatica, e vendette UNIX a Novell nel 1993; poi Novell lo vendette a Santa Cruz Operation nel 1995, ma ormai non aveva più molta importanza chi possedesse UNIX, visto che molte altre società possedevano la licenza.

## 10.1.4 UNIX di Berkeley

Una delle tante università che per prima acquistò la Versione 6 di UNIX fu l'Università della California di Berkeley. Poiché era disponibile il codice sorgente completo, a Berkeley ebbero la possibilità di modificare il sistema in modo sostanziale. Aiutati da contributi ricevuti da ARPA, la Advanced Research Projects Agency del Dipartimento della Difesa americano, a Berkeley produssero e distribuirono una versione migliore per il PDP-11 chiamata **1BSD** (First Berkeley Software Distribution). A questa seguì velocemente la **2BSD**, sempre per il PDP-11.

Più importanti furono la **3BSD** e specialmente il suo successore, il **4BSD**, per il VAX. Sebbene la AT&T avesse una versione di UNIX per il VAX, chiamata **32V**, si trattava, sostanzialmente, della versione 7. Al contrario il **4BSD** (che comprende il **4.1BSD**, il **4.2BSD**, il **4.3BSD** e il **4.4BSD**) conteneva un notevole numero di miglioramenti; il più importante era l'utilizzo della memoria virtuale e la paginazione, che permettevano l'uso di programmi più grandi della memoria fisica suddividendoli in pagine che potevano essere caricate e scaricate dalla memoria secondo le necessità. Inoltre si potevano usare nomi di file più lunghi di 14 caratteri. Anche il file system fu modificato rendendolo più veloce e la gestione dei segnali fu resa più affidabile. Fu introdotta anche la possibilità di collegamento in rete tramite un protocollo di rete, il **TCP/IP**, che divenne uno standard de facto nel mondo UNIX, e più tardi in Internet, che è dominata dai server UNIX.

A Berkeley furono anche aggiunti una buona quantità di programmi di utilità a UNIX, compreso un nuovo editor (**vi**) e una nuova shell (**csh**), i compilatori Pascal e Lisp e molti altri. Tutti questi miglioramenti furono la causa per cui la Sun Microsystem, la DEC e altre case venditrici di computer basarono le loro versioni di UNIX su quella dell'Università di Berkeley piuttosto che su quella ufficiale della AT&T, System V. Lo UNIX di Berkeley diventò ben noto nel mondo accademico, delle ricerche e della difesa. Per avere maggiori informazioni sullo UNIX di Berkeley si guardi McKusick e al., 1996.

## 10.1.5 UNIX standard

Dalla fine degli anni '80 erano largamente in uso due versioni differenti e considerabilmente incompatibili di UNIX, la **4.3BSD** e la **Release 3** di **System V**. Inoltre quasi ogni venditore aggiunse i suoi miglioramenti non standard. Questa divisione nel mondo di UNIX, insieme con la mancanza di formati standard per il programma binario impediti il successo commerciale perché per i venditori era impossibile scrivere e consegnare programmi UNIX aspettandosi che essi avrebbero girato su qualsiasi sistema UNIX (come era normale per MS-DOS). Inizialmente fallirono diversi tentativi di standardizzazione di UNIX. Per esempio, la AT&T mise a punto il **SVID** (System V Interface Definition), che definiva le chiamate di sistema, i formati dei file, eccetera per cercare di tenerli in linea tutti i vendori di **System V**, ma questo tentativo non ebbe alcun effetto nel mondo nemico (**BSD**) dove si limitarono ad ignorarlo.

Il primo tentativo serio di unificare le due versioni di UNIX fu iniziato grazie all'intervento del IEEE Standards Board, una corporazione largamente rispettata, molto importante e neutrale. Alla realizzazione di questo lavoro parteciparono centinaia di persone provenienti dall'industria, dall'università ed anche dal governo. Il nome dato collettivamente a questo progetto è **POSIX**, dove le prime tre lettere significano Portable Operating System (Sistema Operativo Portatile) e la **IX** fu aggiunta per ottenere un nome simile a **UNIX**.

Dopo diverse discussioni, proposte e controposte, il comitato di **POSIX** produsse una versione standard conosciuta con il nome **1003.1**. Essa definisce un insieme di pro-

cedure di libreria, che devono essere supportate dai sistemi conformi. La maggior parte di queste procedure effettua una chiamata di sistema, ma qualcuna è implementata fuori dal kernel; procedure tipiche sono *open*, *fork* e *read*. L'idea base di POSIX è che un venditore di software che scrive un programma usando solo le procedure definite in 1003.1 sa che esso girerà su qualunque versione standard di UNIX.

Sebbene sia vero che la maggior parte dei comitati di standardizzazione tenda a produrre un orribile compromesso con una piccola parte delle caratteristiche preferite di ciascuno, la versione 1003.1 prende in considerazione tutti gli interessi coinvolti in questa situazione, producendo un risultato decisamente buono. Il comitato IEEE, invece di considerare come punto di partenza l'unione di System V e BSD (come fanno di solito i comitati di standardizzazione), ne prese l'intersezione. In modo molto grossolano, nella versione standard furono incluse solo quelle parti presenti sia nella versione di System V sia di BSD, e la conseguenza è che 1003.1 assomiglia molto al diretto predecessore di System V e BSD, cioè Version 7. Le due aree in cui differisce maggiormente dalla Version 7 sono i segnali (che provengono soprattutto da BSD) e la gestione del terminale, che è nuova. Il documento su 1003.1 è scritto in modo tale che sia i realizzatori di sistemi operativi che i programmati possano capirlo, un'altra novità nel mondo degli standard, sebbene si stia già lavorando per eliminare questa caratteristica.

Sebbene la versione standard di 1003.1 tratti solo le chiamate di sistema, i documenti ad esso relativi standardizzano i programmi di utilità, le reti e molte altre parti di UNIX. C'è da aggiungere che il linguaggio C è stato standardizzato anche da ANSI e ISO.

Sfortunatamente, al rientro da un incontro sugli standard successe una cosa divertente: ora che si erano unificate la versione System V e quella BSD, ne comparve un'altra. Ad un gruppo di vendori come IBM, DEC, Hewlett-Packard e molti altri non piacque l'idea che la AT&T avesse il controllo di UNIX, così misero su un consorzio chiamato OSF (Open Software Foundation) per produrre un sistema che comprendesse gli standard di IEEE e degli altri, ma che in aggiunta contenesse un gran numero di ulteriori possibilità come un sistema di finestre (X11), un'interfaccia utente grafica (MOTIF), un calcolo distribuito (DCE), una gestione distribuita (DME) ed ancora altre cose.

Come reazione la AT&T costituì un altro consorzio, lo UI (UNIX International) per fare esattamente le stesse cose. La versione di UNIX di UI è basata sul System V. Il risultato è che attualmente ci sono due potenti gruppi di industrie che offrono il loro prodotto UNIX, così gli utenti sono lontani dall'avere uno standard unico. Ad ogni modo il mercato decise che System V era migliore di OSF, così quest'ultimo lentamente scomparì. Alcune aziende hanno la propria versione UNIX, come la Solaris della Sun (basata su System V).

## 10.1.6 MINIX

Una proprietà che tutti questi sistemi hanno è che essi sono grandi e complicati, in un certo senso, in antitesi con l'idea originale che è alla base di UNIX. Anche se il codice è disponibile gratuitamente, il che non avviene spesso, non è pensabile che una sola persona lo possa capire. Questa situazione ha portato l'autore di questo libro a scrivere un nuovo sistema tipo-UNIX piccolo abbastanza da essere capito, disponibile come codice sorgente e che può essere utilizzato per scopi di insegnamento. Quel sistema consisteva in 11800 righe di codice C ed 800 righe di codice assembler. Uscì nel 1987, e fu da un punto di vista funzionale paragonabile a UNIX versione 7, il punto di base dei dipartimenti di informatica durante l'era dei PDP-11.

MINIX fu uno dei primi sistemi tipo-UNIX basato su un progetto che utilizza un microkernel. L'idea dietro un microkernel è quella di fornire minime funzionalità nel ker-

nel per renderlo efficiente e affidabile. Di conseguenza, la gestione della memoria e il file system furono posti fuori come processi utente. Il kernel era di 1600 righe di codice C e 800 righe di codice assembler. Per ragioni tecniche legate all'architettura dell'8088, i driver di dispositivo I/O (2900 righe aggiuntive in C) si trovavano sempre nel kernel. Il file system (5100 righe di C) ed il gestore della memoria (2200 righe di C) giravano invece come due processi utente separati.

I microkernel hanno il vantaggio, rispetto al sistema monolitico, di essere facili da capire e facili da mantenere, visto la modularità della loro struttura. Inoltre, il fatto di spostare il codice fuori dal kernel, rende i sistemi più affidabili poiché il crash di un processo utente non intacca l'intero sistema. Lo svantaggio maggiore è un leggero abbassamento delle prestazioni dovuto ad ulteriori commutazioni da modo kernel a modo utente e viceversa. Ad ogni modo, le prestazioni non sono tutto: tutti i moderni sistemi UNIX eseguono X Windows in modo utente ed accettano che le prestazioni diminuiscano leggermente a favore della modularità (in contrasto con Windows, dove l'intera GUI è nel kernel). Altri sistemi a struttura a microkernel famosi dell'era MINIX erano Mach (Accetta et al., 1986), e Chorus (Rozier et al., 1988). Una completa discussione sulle prestazioni dei sistemi a microkernel è riscontrabile in (Bricker et al., 1991).

Dopo pochi mesi dalla sua comparsa, MINIX divenne una specie di cult, con il proprio newsgroup, *comp.os.minix*, ed oltre 40000 utenti. Molti utenti aiutarono nello sviluppo di comandi e programmi utente, cosicché MINIX divenne in breve una impresa collettiva fatta da un ampio numero di persone in Internet: fu il prototipo di altri sviluppi collettivi che vennero in seguito. Nel 1997, la versione 2.0 di MINIX uscì, ed il sistema di base, che adesso conteneva anche le problematiche di rete, era cresciuto fino a 62200 righe di codice. Un libro sui sistemi operativi, spiegati attraverso il codice di 500 pagine di MINIX (dato in appendice e nel CD-ROM di accompagnamento) è Tanenbaum e Woodhull, 1997. MINIX è disponibile anche in rete gratuitamente all'indirizzo [www.cs.vu.nl/~ast/minix.html](http://www.cs.vu.nl/~ast/minix.html).

## 10.1.7 Linux

Durante i primi anni dello sviluppo di MINIX e delle discussioni in Internet, molte persone richiesero (ed in molti casi, pretesero) caratteristiche più precise e migliori, alle quali l'autore spesso disse "NO" (per mantenere il sistema abbastanza piccolo da renderlo comprensibile agli studenti in un corso semestrale universitario). Questo continuo "NO" urtò molti utenti. In quel preciso momento, FreeBSD non era disponibile, quindi non vi era molta scelta. Dopo che per alcuni anni le cose rimasero così, uno studente finlandese, Linus Torvalds, decise di scrivere un altro clone di UNIX, chiamato Linux che doveva diventare un sistema operativo completo di mille caratteristiche cosa che MINIX (volutamente) non era. La prima versione di Linux, 0.01, uscì nel 1991. Fu sviluppata su una macchina MINIX ed ereditava alcune idee da MINIX a partire dalla struttura dell'albero sorgente fino al layout del file system. Ad ogni modo, esso era un sistema monolitico piuttosto che un sistema a microkernel, con l'intero sistema operativo presente nel kernel stesso. Il codice totale era di 9300 righe di codice C e 950 righe di codice assembler, molto simile alla versione MINIX in dimensione e grossolanamente comparabile in funzionalità.

Linux crebbe velocemente e si trasformò in un prodotto clone completo di UNIX con memoria virtuale, un file system sofisticato, e molte altre caratteristiche. Sebbene esso girava originariamente solo su 386 (ed aveva anche codice 386 in mezzo a procedure C), fu velocemente portato su altre piattaforme ed, oggi, gira su una ampia varietà di macchine, esattamente come fa UNIX. Esiste comunque una notevole differenza tra Linux e

UNIX: Linux fa grande uso di molte caratteristiche che possiede il compilatore `gcc`, e necessita di molto lavoro prima di poter essere compilato con un compilatore ANSI C standard senza caratteristiche aggiuntive.

La più importante release di Linux è sicuramente la versione 1.0, uscita nel 1994. Essa consisteva in circa 165000 righe di codice ed includeva un nuovo file system, file mappati in memoria, e una rete BSD - compatibile con socket e TCP/IP. Linux includeva anche molti driver di dispositivi; nei due anni successivi seguirono poi parecchie revisioni di importanza minore.

A questo punto, Linux era sufficientemente compatibile con UNIX e quindi un certo quantitativo di software per UNIX venne portato a Linux, rendendolo molto più utile di quanto non potesse esserlo senza. In aggiunta, molte persone furono molto interessate a Linux ed incominciarono a lavorare sul codice ed ad estenderlo in molti modi sotto la supervisione di Torvalds.

La release successiva, la 2.0, fu resa pubblica nel 1996. Essa si componeva di circa 470000 righe di C ed 8000 righe di assembler, con un supporto per architetture 64-bit, la multiprogrammazione simmetrica, nuovi protocolli di rete, e numerose altre caratteristiche; gran parte del codice totale era data da una collezione di driver di dispositivi. Release addizionali seguirono, quindi, in modo frequente.

Un ampio insieme di software standard per UNIX è stato trasportato a Linux, compresi più di 1000 programmi di utilità, X Windows e molto software legato alla rete. Due GUI diverse (GNOME e KDE) sono state scritte anche per Linux. In breve, Linux è cresciuto fino ad essere un clone completo di UNIX con tutti gli accessori che un amante di UNIX può desiderare.

Una caratteristica inusuale di Linux è il suo modello di distribuzione: è un software gratuito. Esso può essere scaricato da vari siti su Internet, per esempio: [www.kernel.org](http://www.kernel.org). Linux è accompagnato da una licenza decisa da Richard Stallman, fondatore della Free Software Foundation. Benché Linux sia gratuito, la sua licenza, la licenza **GPL (GNU Public License)**, è ben più lunga e complessa della licenza di Windows 2000 e specifica in modo preciso cosa può essere fatto con il software e con il codice: gli utenti possono usare, copiare, modificare e ridistribuire il sorgente ed il codice binario gratuitamente. La maggiore restrizione è che tutto il lavoro derivato dal kernel Linux non può essere venduto o ridistribuito in forma solo binaria; il codice sorgente deve seguire il prodotto o essere recuperabile se richiesto.

Sebbene Torvalds controlli ancora il kernel in modo abbastanza preciso, altri programmati hanno scritto molto software a livello utente, e molti di essi sono migrati da comunità di software on-line quali MINIX, BSD e GNU. Ad ogni modo, durante l'evolversi di Linux, solo una piccola frazione della comunità Linux decise di trafficare con i sorgenti (testimoni le centinaia di libri che narrano come installare ed usare Linux e solo poche discussioni sul codice vero e su come funziona). Inoltre, molti utenti Linux preferiscono acquistare un CD-ROM dalle tante aziende che lo vendono piuttosto che scaricare il software via Internet (Un sito web dove trovare più di 50 società che vendono differenti pacchetti Linux è [www.linux.org](http://www.linux.org)). Con l'aumentare delle società che vendono varie versioni personali di Linux ed il crescere delle aziende che decidono di preinstallare il software Linux sulle macchine che vendono, la linea di distinzione tra software commerciale e software gratuito incomincia a non essere più così netta.

Come postilla alla storia di Linux, è interessante notare che proprio mentre il treno Linux stava accelerando, una forte spinta gli venne da una direzione inaspettata - l'AT&T. Nel 1992, Berkeley decise di terminare per mancanza di fondi lo sviluppo di BSD con una release finale, 4.4BSD, (che più tardi rappresentò la base per FreeBSD). Poiché questa versione praticamente non conteneva codice AT&T, Berkeley mise il software sotto una licenza gratuita (non GPL) che permetteva a tutti di operare sul software purché non inten-

tassero causa all'Università della California. L'AT&T, che controllava UNIX, reagì - come potete immaginare - prontamente, perseguiendo legalmente l'Università della California, e contemporaneamente una società, BSDI, che era stata fondata dai progettisti BSD in modo da supportare e vendere il sistema (più o meno come Red Hat ed altre aziende fanno adesso per Linux). Poiché virtualmente non era in gioco alcun codice AT&T, la battaglia legale fu su copyright e marchi, come ad esempio il numero di telefono della BSDI 1-800-ITS-UNIX. Sebbene il caso finisse con un accordo extragiudiziale, questa azione legale mantenne FreeBSD fuori dal mercato abbastanza perché Linux potesse diventare un sistema stabile e conosciuto. Se la battaglia legale non fosse avvenuta, dal 1993 ci sarebbero stati sul mercato due sistemi UNIX gratuiti e open-source, e tra di loro una notevole competizione: BSD, un sistema maturo e stabile con una lunga schiera di interventi accademici sin dal 1977, contro un vigoroso e giovane contendente, Linux, vecchio solo di due anni ma che stava crescendo tra gli utenti individuali. Chi può sapere come si sarebbe conclusa questa battaglia tra due UNIX gratuiti?

Data questa storia, la stretta conformità a POSIX, e la sovrapposizione tra le due comunità di utenti, non dovrebbe essere una sorpresa il fatto che molte caratteristiche di Linux, le chiamate di sistema, i programmi, le librerie, gli algoritmi, e le strutture dati interne sono molto simili a quelle di UNIX. Per esempio, più dell'80% delle circa 150 chiamate di sistema Linux sono copia esatta delle corrispondenti chiamate di sistema in POSIX, BSD o System V. Quindi, come prima approssimazione, la maggior parte delle descrizioni di UNIX, date in questo capitolo, possono essere considerate anche per il caso Linux. Nel caso in cui vi siano sostanziali differenze, (ad esempio l'algoritmo di scheduling) tra UNIX e Linux, cercheremo di evidenziare e coprire entrambe le soluzioni. Dove Linux e UNIX sono essenzialmente identici, ci riferiremo semplicemente a UNIX, per brevità nella trattazione. Avvertiamo che Linux sta evolvendo velocemente, e con migliaia di persone che stanno trafficando sul sorgente, un po' del materiale trattato (basato sulla versione 2.2) potrebbe diventare presto obsoleto.

## 10.2 Panoramica di UNIX

In questo paragrafo si darà un'introduzione generale su UNIX e sul suo utilizzo per aiutare i lettori che non lo conoscono. Sebbene versioni diverse di UNIX differiscano tra loro in modo subdolo, il materiale qui presentato si potrà applicare a tutte. L'attenzione in questa parte è su come UNIX appare agli utenti finali; nelle sezioni successive ci concentreremo sulle chiamate di sistema e su come il sistema stesso funziona al suo interno.

### 10.2.1 Obiettivi di UNIX

UNIX è un sistema interattivo pensato per gestire processi multipli ed utenti multipli simultaneamente. È stato progettato da programmati, per programmati, da usare in un ambiente in cui la maggior parte degli utenti sono relativamente sofisticati e sono coinvolti nello sviluppo di progetti software (spesso molto complicati). In molti casi un gran numero di programmati cooperano attivamente per produrre un singolo sistema, per cui UNIX dà agli utenti molte facilitazioni per lavorare insieme e per condividere le informazioni in maniera controllata. Il modello di un gruppo di programmati esperti che lavorano insieme per produrre software avanzato è chiaramente molto diverso dal modello del personal computer di un principiante che lavora da solo con un programma di video scrittura, e questa differenza si nota in UNIX dall'inizio alla fine.

Cosa vuole un buon programmatore da un sistema? Per iniziare, i più vorrebbero un sistema semplice, elegante e coerente. Per esempio, a livello più basso, un file sarà semplicemente un insieme di byte, per cui avere diverse classi di file per l'accesso sequenziale, l'accesso casuale, l'accesso con chiave, l'accesso remoto (come capita sui mainframe) è solo un impiccio. Allo stesso modo, se il comando

`ls A*`

significa elenca tutti i file che iniziano con "A" allora il comando

`rm A*`

significherà rimuovi tutti i file che iniziano con "A" e non semplicemente rimuovi il file il cui nome è formato da una "A" più un asterisco. Questa caratteristica talvolta è chiamata il *principio di minima sorpresa (principle of least surprise)*.

Un'altra cosa che i programmatore esperti generalmente richiedono è la flessibilità e la potenza. Ciò significa che un sistema dovrà avere un piccolo numero di elementi di base che si possono combinare in una varietà infinita di modi per adeguare l'applicazione. Un'idea base dietro UNIX è che ogni programma dovrà fare solo una cosa e dovrà farla bene, per cui i compilatori non producono listati perché altri programmi possono farlo meglio.

Infine molti programmatore hanno una forte avversione per la ridondanza inutile, per esempio, perché digitare *copy* quando basta *cp*? Per ottenere tutte le righe che contengono la stringa "ard" dal file *f*, il programmatore UNIX digita

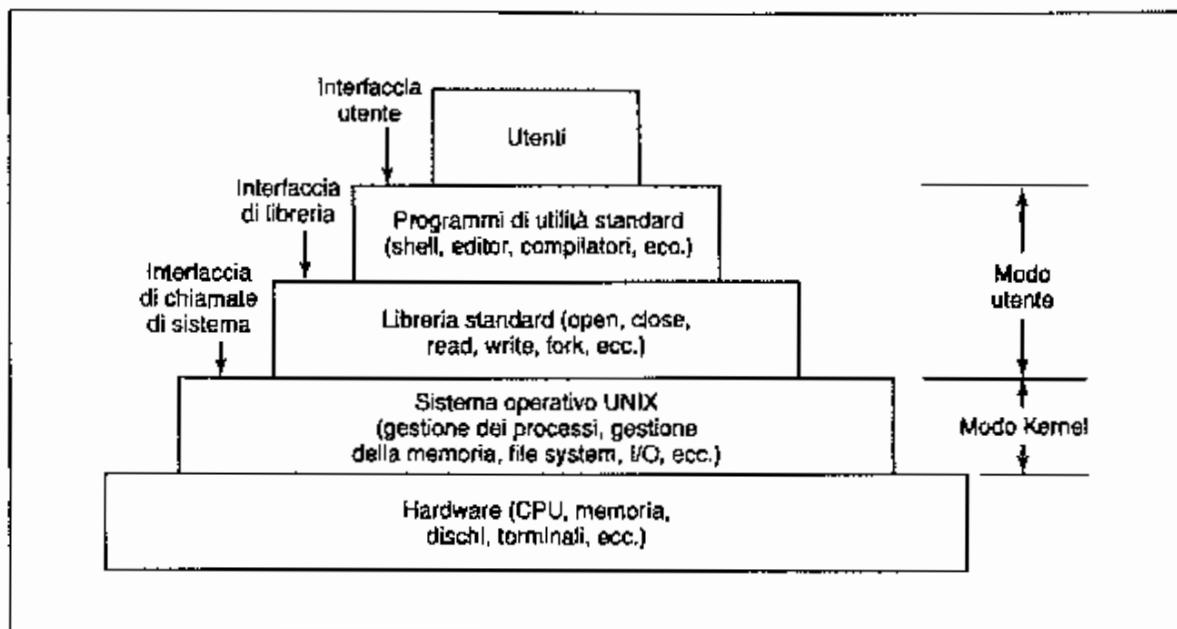
`grep ard f`

L'approccio contrario è che il programmatore prima seleziona il programma *grep* (senza argomenti) e poi vede il messaggio: "Ciao, sono *grep*, aspetto gli argomenti. Per favore, inseriscili". Dopo aver ottenuto gli argomenti *grep* attende il nome del file e chiede se ci sono altri nomi di file; poi scrive quello che sta per fare e chiede se è corretto. Mentre questo modo di procedere può essere ben accetto ai principianti, di fatto annoia gli esperti che vogliono un esecutore attivo e non una balia.

## 10.2.2 Interfacce con UNIX

Un sistema UNIX si può pensare formato come una piramide, come illustrato in Figura 10.1. Al livello più basso si trova l'hardware, formato dalla CPU, memoria, dischi, terminali ed altre periferiche ed è su questa parte che gira il sistema operativo UNIX. La sua funzione è controllare l'hardware e fornire a tutti i programmi un'interfaccia con le chiamate di sistema, le quali permettono ai programmi utenti di creare e gestire processi, i file e le altre risorse.

I programmi effettuano le chiamate di sistema mettendo gli argomenti nei registri (o qualche volta in uno stack) ed eseguendo un'istruzione di trap per passare dallo stato utente allo stato kernel, facendo partire UNIX. Poiché, in C, non c'è modo di scrivere un'istruzione di trap, viene fornita una libreria con una procedura per ogni chiamata di sistema; queste procedure sono scritte in linguaggio assembler, ma si possono chiamare da C. Per prima cosa ognuna di esse mette gli argomenti nel posto appropriato e poi esegue l'istruzione di trap, così, per eseguire la chiamata di sistema *read*, un programma C chiama la procedura di libreria *read*. In effetti, è l'interfaccia con la libreria e non l'interfaccia



**Figura 10.1** Gli strati del sistema UNIX.

con le chiamate di sistema che è specificata da POSIX, cioè POSIX stabilisce quali procedure di libreria deve fornire un sistema, quali sono i loro parametri, cosa devono fare e quali risultati devono restituire, e non fa nessuna menzione delle chiamate di sistema.

Oltre al sistema operativo e alla libreria di chiamate di sistema, tutte le versioni di UNIX forniscono diversi programmi standard, alcuni dei quali sono specificati da POSIX standard 1003.2, mentre altri sono differenti tra le varie versioni UNIX. Fra essi ci sono l'interprete dei comandi (shell), i compilatori, i programmi che elaborano il testo e i programmi di utilità che gestiscono i file. Tutti questi programmi possono essere chiamati direttamente dall'utente.

È pertanto possibile parlare di tre diverse interfacce UNIX: l'interfaccia vera e propria basata sulle chiamate di sistema, la libreria di interfaccia e l'interfaccia formata dall'insieme dei programmi standard di utilità. Questi ultimi, che sono percepiti dall'utente finale come parte di UNIX, in realtà non hanno quasi niente a che fare con il sistema operativo e possono essere facilmente rimpiazzati.

Alcune versioni di UNIX, per esempio, hanno sostituito l'interfaccia utente per la tastiera con un'interfaccia utente grafica per mouse, senza cambiare minimamente il sistema operativo. È esattamente questo tipo di flessibilità che rende UNIX così popolare e gli ha permesso di sopravvivere così bene all'evoluzione tecnologica.

### 10.2.3 La shell di UNIX

Molti sistemi UNIX hanno una interfaccia grafica utente simile ai Macintosh ed a Windows, ma i veri programmatore preferiscono ancora una interfaccia a righe di comando, la shell. È molto più veloce da utilizzare, più potente, facilmente estendibile, e non favorisce la comparsa di dolori al polso, causati all'utente dal dover usare il mouse in ogni momento. Di seguito, descriveremo brevemente la shell di Bourne (*sh*) ma da allora sono state scritte molte shell (*ksh*, *bash*, eccetera). Sebbene UNIX supporti un ambiente completamente grafico (X Windows), in esso i programmatore aprono diverse shell ed agiscono come se avessero una dozzina di terminali ASCII aperti, ognuno con una shell diversa.

La shell inizializza se stessa, poi emette un carattere di **prompt** (attesa), spesso il carattere percentuale ("%") o il carattere dollaro ("\$"), e attende che l'utente batte una linea di comando.

Quando l'utente batte una linea di comando, la shell estrae da essa la prima parola, suppone che questa sia il nome di un programma che deve essere chiamato, lo cerca, e se lo trova lo manda in esecuzione. A questo punto, si autosospende finché il programma non termina, e poi passa a leggere il comando successivo. È importante notare che la shell è semplicemente un programma utente come gli altri: il suo compito è quello di leggere e scrivere sul terminale, e poter mandare in esecuzione altri programmi.

I comandi possono avere degli argomenti, che sono passati al programma chiamato come stringhe di caratteri. Per esempio, la linea di comando

```
cp src dest
```

chiama il programma *cp* con due argomenti, *src* e *dest*. Questo programma interpreta il primo come il nome di un file già esistente; fa una copia di questo file e chiama la copia *dest*.

Non tutti gli argomenti sono nomi di file. In

```
head -20 file
```

il primo argomento, -20, chiede a *head* di stampare le prime 20 linee di *file*, invece del numero considerato per difetto che è 10 linee. Gli argomenti che controllano l'esecuzione di un comando o che specificano un valore opzionale sono detti **flag** (indicatori) e per convenzione sono preceduti dal carattere meno. Questo carattere è necessario per evitare ambiguità, perché il comando

```
head 20 file
```

è perfettamente legale in quanto chiede a *head* di stampare le prime 10 linee di un file chiamato 20 e, successivamente, le prime 10 linee di un secondo file chiamato *file*. La maggior parte dei programmi UNIX accetta numerosi flag e argomenti.

Per facilitare la specifica dei nomi dei file, la shell accetta **caratteri magici** (**magic character**), detti talvolta **metacaratteri** (**wild card**). Per esempio, un asterisco corrisponde a tutte le possibili stringhe, per cui

```
ls *.c
```

chiede a *ls* di elencare tutti i file i cui nomi terminano con *.c*. Se esistono i file chiamati *x.c*, *y.c* e *z.c*, il comando precedente corrisponde a battere

```
ls x.c y.c z.c
```

Un altro metacarattere è il punto di domanda che corrisponde a un qualsiasi carattere. Una lista di caratteri dentro la parentesi quadra seleziona uno di essi, cosicché

```
ls {ape}*
ls {ape}*
```

elena tutti i file che incominciano con "a", "p", o "e".

Un programma di shell non deve aprire il terminale per leggere da esso o per scriverci sopra. Al contrario, quando esso (o qualsiasi altro programma) parla, automaticamente ha accesso ad un file chiamato **standard input** (per la lettura), ad un file chiamato **stan-**

**standard output** (per la scrittura) e ad un file chiamato **standard error** (per scrivere i messaggi di errore). Di solito questi tre file sono per difetto associati al terminale, per cui una lettura dallo standard input di fatto legge dalla tastiera, e una stampa sullo standard output o sullo standard error compare sul video. Molti programmi UNIX leggono dallo standard input e scrivono sullo standard output per difetto. Per esempio

```
sort
```

chiama il programma *sort* che legge le linee dal terminale (finché l'utente non digita un CTRL-D per indicare la fine del file) le ordina alfabeticamente e scrive il risultato sul video.

È possibile e spesso utile ridirigere (redirect) lo standard input e lo standard output. La sintassi per ridirigere lo standard input usa il segno minore (<) seguito dal nome del file. Allo stesso modo, lo standard output si ridirige con il segno maggiore (>). Si possono usare entrambi i simboli sullo stesso comando. Per esempio, il comando

```
sort <in >out
```

dice a *sort* di prendere il suo input dal file *in* e di scrivere il suo output sul file *out*. Poiché lo standard error non può essere ridiretto, qualunque messaggio di errore va sul video. Un programma che legge il suo input dallo standard input, fa qualche calcolo su di esso e scrive il suo output sullo standard output è chiamato **filtro**.

Si consideri la seguente linea di comando

```
sort <in >temp; head -30 <temp; rm temp
```

Per primo gira il programma *sort*, prendendo l'input da *in* e scrivendo l'output su *temp*. Completato questo lavoro, la shell fa partire *head*, dicendogli di stampare le prime 30 linee di *temp* e le stampa sullo standard output, che è per difetto il terminale; infine il file temporaneo viene rimosso.

Accade frequentemente che il primo programma in una linea di comando produca un output che è usato come input del programma successivo. Nell'esempio precedente si è usato il file *temp* per questo scopo, ma UNIX fornisce un modo più semplice per fare la stessa cosa. In

```
sort <in | head -30
```

la linea verticale, chiamata **pipe symbol** (simbolo di connessione), dice di prendere l'output da *sort* ed usarlo come input di *head*, eliminando la necessità della creazione, l'utilizzo e la rimozione del file temporaneo. Un insieme di comandi collegati dai simboli di pipe, chiamata **pipeline**, può contenere un numero arbitrario di comandi. Una pipeline di quattro componenti è mostrato nell'esempio che segue:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Qui tutte le linee che contengono la stringa "ter" in tutti i file che terminano con *.t* sono scritte sullo standard output, dove vengono ordinati. Le prime 20 righe sono selezionate da *head* che le passa a *tail*, che ne scrive le ultime cinque (le linee dalla 16 alla 20 nella lista ordinata) sul file *foo*. Questo è un esempio di come UNIX fornisce blocchi essenziali (numerosi filtri), ognuno dei quali fa un lavoro, con un meccanismo che permette di metterli insieme in modo quasi del tutto illimitato.

UNIX è un sistema multiprogrammato general purpose, dove un singolo utente può fare girare più programmi contemporaneamente, ognuno con un processo separato. La sintassi di shell per lanciare un processo in background è quella di far seguire il comando da una "e" commerciale (&), per cui

```
wc -l <a >b &
```

fa girare il programma *wc* (word count, contatore di parole) per contare il numero di linee (flag *-l*) del suo input, *a*, scrivendo il risultato su *b*, ma lavorando in background. Appena il comando è stato digitato, la shell manda il prompt ed è pronta ad accettare e a gestire il prossimo comando. Le pipeline possono essere usate anche in background, per esempio

```
sort <x | head 5
```

Più pipeline possono girare contemporaneamente in background.

È possibile mettere un elenco di comandi shell in un file e poi lanciare una shell con questo file come standard input; la seconda shell si limita ad eseguirli in ordine, come se fossero digitati uno dopo l'altro da tastiera. I file che contengono comandi shell sono detti *shell script* (programmi di shell); essi possono assegnare valori alle variabili di shell e poi leggerli in un secondo momento, e possono anche avere dei parametri ed usare dei costrutti *if*, *for*, *while* e *case*. Si deduce che uno *shell script* è realmente un programma scritto in linguaggio shell. La shell C di Berkeley è una shell alternativa che è stata progettata per scrivere shell script molto simili ai programmi C, così come assomiglia al C il suo linguaggio di comandi. E siccome le shell sono programmi utente, anche molte altre persone hanno scritto e distribuito delle shell.

## 10.2.4 Programmi di utilità di UNIX

L'interfaccia utente di UNIX non è formata solo dalla shell ma anche da un considerevole numero di programmi standard di utilità. Si può fare una suddivisione grossolana di questi programmi in sei categorie:

1. Comandi di gestione file e directory.
2. Filtri.
3. Compilatori e strumenti per lo sviluppo di programmi.
4. Trattamento dei testi.
5. Amministrazione di sistema.
6. Altri.

Lo standard POSIX 1003.2 specifica la sintassi e la semantica di un centinaio di questi programmi, soprattutto nelle prime tre categorie. L'idea di standardizzarli è quella di rendere possibile la scrittura di script shell che usano questi programmi e che funzionano su qualunque sistema UNIX. In aggiunta a queste utilità standard, vi sono anche molti altri programmi come, ad esempio, browser per il Web, visualizzatori di immagini, eccetera.

Si consideri qualche esempio, iniziando con la manipolazione di file e directory.

```
cp a b
```

copia il file *a* su *b*, lasciando il file *a* intatto. Al contrario

`mv a b`

copia *a* su *b* ma elimina l'originale; di fatto, questo programma sposta (cambia il nome) il file piuttosto che farne una copia in senso stretto. Molti file possono essere concatenati usando *cat* che legge i suoi file in input e li copia tutti sullo standard output, uno dopo l'altro. I file possono essere cancellati con il comando *rm*. Il comando *chmod* permette al proprietario di cambiare i permessi di accesso al file. Le directory si possono creare con il comando *mkdir* e cancellare con *rmdir*. Il comando *ls* permette di vedere l'elenco dei file appartenenti alla directory. Ci sono diversi tipi di flag, per controllare tutti i possibili attributi di ogni file (per esempio dimensione, proprietario, gruppo, data di creazione), per determinare il tipo di ordinamento (per esempio alfabetico, la data dell'ultima modifica), per specificare il formato del video e molti altri.

Si sono già visti diversi filtri: *grep* estrae le linee che contengono una data configurazione dallo standard input o da uno o più file; *sort* ordina il suo input e lo scrive sullo standard output; *head* estrae le linee iniziali dal suo input; *tail* estrae le linee terminali del suo input. Nella versione 1003.2 sono definiti altri filtri come *cut* e *paste*, che permettono di prendere le colonne di un testo e di portarle da un'altra parte del file; *od*, che converte il suo input (di solito in binario) in un testo in codice ASCII, ottale, decimale o esadecimale; *tr*, che trasforma il carattere (per esempio da minuscolo a maiuscolo) e *pr*, che prepara l'output per la stampante includendo alcune opzioni per aggiungere le intestazioni di pagina, il numero di pagina e così via.

Compilatori e strumenti di programmazione comprendono *cc*, che chiama il compilatore C, e *ar*, che raccoglie le procedure di libreria in file di archivio.

Un altro strumento importante è *make*, che si usa per mantenere grossi programmi il cui codice sorgente è formato da numerosi file. Alcuni di questi sono file di intestazioni (*header*), che contengono tipi, variabili, macro ed altre dichiarazioni; i file sorgenti spesso li includono con l'istruzione speciale *include*, così, più file sorgenti possono condividere le stesse dichiarazioni. Ad ogni modo, se un header file è modificato, è necessario trovare tutti i file sorgenti che dipendono da esso e ricompilarli: la funzione di *make* è quella di tenere traccia di quali file dipendono da un certo file header (ma non solo questo) e fare in modo che per tutti questi la compilazione sia fatta automaticamente. Praticamente tutti i programmi UNIX, eccetto i più piccoli, sono predisposti per essere compilati con *make*.

Una selezione dei programmi di utilità POSIX è elencata in Figura 10.2, accompagnata da una breve descrizione per ognuno. Tutti i sistemi UNIX sono dotati di questi e di molti altri programmi.

## 10.2.5 Struttura del kernel

In Figura 10.1 abbiamo visto la struttura di insieme del sistema UNIX. Cerchiamo adesso di guardare più da vicino il kernel prima di esaminare le altre parti. Identificare una generica struttura del kernel non è così semplice, vista l'esistenza di molte versioni di UNIX, ma, sebbene il diagramma di Figura 10.3 si riferisca a 4.4BSD, esso può essere applicato a molte altre versioni con piccoli cambiamenti qui e là.

Lo strato di base del kernel è formato da driver di dispositivi e dallo smistamento dei processi. I driver UNIX possono essere classificati come driver di dispositivi di carattere o driver di dispositivi di blocco, con la principale differenza che il posizionamento (seek) è permesso solo sui dispositivi a blocco e non sui dispositivi a carattere. Tecnicamente, i driver di rete sono driver di carattere, ma vengono gestiti in modo così speciale che è più chiaro, probabilmente, tenerli separati dal resto, come è stato fatto in figura. Lo smistamento dei processi avviene al momento di un'interruzione: il codice a basso livello non

| Programma | Uso tipico                                    |
|-----------|-----------------------------------------------|
| cat       | Concatena file multipli nello standard output |
| chmod     | Cambia il livello di protezione dei file      |
| cp        | Copia uno o più file                          |
| cut       | Elimina colonne di testo dal file             |
| grep      | Ricerca nel file dato un certo pattern        |
| head      | Estrae le prime righe da un file              |
| ls        | Elenco il contenuto della directory           |
| make      | Compila un file per ottenere un binario       |
| mkdir     | Crea una directory                            |
| od        | Dump ottale di un file                        |
| paste     | Incolla colonne di testo in un file           |
| pr        | Formatta un file prima della stampa           |
| rm        | Rimuove uno o più file                        |
| rmdir     | Rimuove una directory                         |
| sort      | Ordina un file per righe in ordine alfabetico |
| tail      | Estrae le ultime righe di un file             |
| tr        | Traduce i codici del carattere                |

Figura 10.2 Una piccola parte dei comuni programmi di utilità UNIX richiesti da POSIX.

| Chiamate di sistema      |                    |                     |                             |                  | Interrupt e trap       |                         |                                      |
|--------------------------|--------------------|---------------------|-----------------------------|------------------|------------------------|-------------------------|--------------------------------------|
| Gestore del terminale    |                    | Socket              | Nominazione di file         | Map-patura       | Page fault             | Gestore di segnali      | Terminazione e creazione di processi |
| Tty grezzo               | Tty cooked         | Protocolli di rete  | File system                 | Memoria virtuale |                        |                         |                                      |
|                          | Discipline di riga | Routing             | Cache di buffer             | Cache di pagine  | Scheduling di processi |                         |                                      |
| Dispositivi di carattere |                    | Dispositivi di rete | Driver di dispositivi disco |                  |                        | Dispatching di processi |                                      |
| Hardware                 |                    |                     |                             |                  |                        |                         |                                      |

Figura 10.3 Struttura del kernel di 4.4BSD.

fa altro che bloccare il processo in esecuzione, e salvare il suo stato nella tabella dei processi del kernel avviando il driver appropriato. Lo stesso avviene anche quando il kernel ha concluso una operazione ed è tempo di iniziare un nuovo processo utente. Il codice per lo smistamento è in assembler ed è separato da quello dedicato allo scheduling.

Sopra il livello di base, il codice risulta diverso in ognuna delle quattro colonne della Figura 10.3. A sinistra, abbiamo i dispositivi di carattere, che possono essere usati in due modi diversi: alcuni programmi, come editor visuali (*vi* e *emacs*), vogliono che ogni lettera sia visualizzata non appena premuta: i terminali raw di I/O (tty) lo rendono possibile. Altri software, come la shell (*sh*), sono orientati ai comandi a linea e permettono agli utenti di

editare la linea corrente prima di premere ENTER e poi spedire il tutto al programma. Questo software utilizza il modo cooked e le discipline di riga.

Il software di rete è spesso modulare, con diversi dispositivi e protocolli. Lo strato superiore ai driver di rete gestisce una funzione di routing che controlla che il pacchetto giusto vada al corretto gestore di dispositivo o di protocollo. La maggior parte dei sistemi UNIX contiene, all'interno del kernel, le funzionalità di un router Internet, sebbene le prestazioni risultino inferiori rispetto ad un router hardware (il codice è però precedente). Al di sopra del codice router vi è lo stack di protocollo, comprendente sempre IP e TCP, ma anche protocolli addizionali. Al di sopra di tutta la rete vi è l'interfaccia socket, che permette ai programmi la creazione di socket per reti particolari e di protocolli, restituendo un descrittore di file, da poter utilizzare in seguito, per ogni socket.

Al di sopra dei driver di dispositivi disco vi sono la cache di buffer del sistema e la cache di pagine. Nei sistemi UNIX iniziali, la cache di buffer era costituita da un pezzo fisso di memoria, che lasciava il resto della memoria a disposizione delle pagine utente. In molti sistemi UNIX moderni, non vi è più un limite fisso ed ogni pagina di memoria può essere utilizzata per entrambe le cache, in base a quale delle due ne ha più bisogno.

Sopra la cache di buffer si trova il file system. La maggior parte dei sistemi UNIX supporta file system multipli, compreso il file system veloce di Berkeley, i file system log-structured, ed altri file system di System V. Tutti condividono la stessa cache di buffer. Al di sopra dei file system si trova la gestione dei nomi dei file, la gestione delle directory, la gestione dei collegamenti simbolici ed hard, ed altre proprietà che sono identiche in ognuno dei file system supportati.

Al di sopra della cache di pagine si trova, invece, il sistema di memoria virtuale, dove avviene tutto il processo di paginazione logica e dove si trova l'algoritmo di sostituzione delle pagine. Quindi, si trova il codice dedicato alla mappatura dei file in memoria virtuale ed il codice dedicato alla gestione dei fault di pagina ad alto livello: è questo il codice atto a scoprire cosa occorre fare quando accade un fault di pagina. Esso prima controlla se il riferimento di memoria è un riferimento valido, quindi dove si trova la pagina richiesta, e come può essere ottenuta.

L'ultima colonna ha a che fare con la gestione dei processi. Sopra il codice che smista i processi si trova lo scheduler, che sceglie quale processo deve essere eseguito al passo successivo. Se i thread vengono gestiti nel kernel, la loro gestione si trova qui, sebbene, in alcuni sistemi UNIX, i thread stessi siano gestiti nello spazio utente. Al di sopra dello scheduler si trova il codice per gestire i segnali, e quello responsabile della spedizione dei segnali stessi alla corretta destinazione, così come il codice di terminazione e creazione di processi.

Lo strato superiore è dedicato all'interfaccia al sistema. Sulla sinistra vi è l'interfaccia di chiamate di sistema; tutte arrivano qui e sono dirette ad uno dei moduli inferiori, in base alla natura della chiamata stessa. Sulla parte destra, invece, vi è l'ingresso delle trap e delle interruzioni, compresi i segnali, i fault di pagina, le eccezioni di processo e le interruzioni di I/O.

## 10.3 Processi in UNIX

Nella sezione precedente si è visto UNIX dalla tastiera, o, in altre parole, ciò che l'autore vede al terminale, dando esempi di comandi shell e programmi di utilità usati frequentemente, quindi si è conclusa la sezione con una breve introduzione alla struttura del sistema. Adesso è tempo di entrare in dettaglio nel kernel e di osservare con più attenzione i concetti di base supportati da UNIX, i processi, la memoria, il file system e l'in-

put/output. Queste nozioni sono importanti perché le chiamate di sistema (l'interfaccia con il sistema operativo) sono responsabili della loro gestione. Per esempio, esistono chiamate di sistema per creare processi, per allocare memoria, per aprire file e per fare operazioni di I/O.

Sfortunatamente, causa l'esistenza di molte versioni di UNIX, esistono delle differenze tra queste; tuttavia, in questo capitolo, cercheremo di enfatizzare le caratteristiche che sono da considerarsi comuni alle varie versioni piuttosto che focalizzare la nostra attenzione su una versione specifica. Per questo, in alcune sezioni (in special modo in quella dedicata all'implementazione), la discussione non può considerarsi valida per tutte le versioni.

### 10.3.1 Concetti di base

Le uniche entità attive in un sistema UNIX sono i processi. I processi UNIX sono molto simili ai processi sequenziali classici visti nel capitolo 2: ogni processo esegue un singolo programma ed ha un solo thread. In altre parole, esso ha un registro contatore di programma (program counter) che tiene traccia dell'indirizzo della prossima istruzione che deve essere eseguita. La maggior parte delle versioni di UNIX permette ad un processo in esecuzione di creare altri thread.

UNIX è un sistema multiprogrammato, per cui processi tra loro diversi e indipendenti possono essere eseguiti allo stesso tempo. Ogni utente può avere diversi processi attivi in un certo momento, per cui su di un grosso sistema ci possono essere centinaia o migliaia di processi in esecuzione. Infatti, su molte workstation a singolo utente, anche quando questi è assente, girano dozzine di processi in background, detti **demoni** (daemon), che partono automaticamente quando il sistema è attivato (in inglese, il termine che indica uno spirito malvagio autonomo si scrive un po' diversamente, "demon" e non "daemon").

Un demone tipico è il **demone cron**, che si attiva ogni minuto per controllare se c'è qualche lavoro da fare, nel qual caso lo svolge, ritornando a fermarsi fino al prossimo controllo.

Questo demone è necessario perché UNIX permette di schedulare le attività future in termini di minuti, ore, giorni e anche mesi. Per esempio, si supponga che un utente abbia un appuntamento con il dentista alle 15 di martedì prossimo. Egli può programmare la base dati del demone cron in modo che il demone emetta un suono di avviso, per esempio, alle 14.30. Quando arriva il giorno e l'ora dell'appuntamento, il demone cron sa che deve svolgere un lavoro per cui fa partire il programma di suono come un nuovo processo.

Il demone cron è usato anche per far partire attività periodiche, tipo il salvataggio giornaliero del disco alle 4 del mattino o per ricordare agli utenti che ogni anno il 31 ottobre devono acquistare i biscotti per Halloween. Altri demoni gestiscono la posta in arrivo e in partenza, le code di stampa, il controllo sulle pagine di memoria disponibili e così via. I demoni sono semplici da implementare perché ognuno di essi è un processo separato e indipendente da tutti gli altri processi.

I processi in UNIX sono creati in un modo particolarmente semplice. La chiamata di sistema fork crea una copia esatta del processo originale, che si chiama **processo genitore** o **padre** (parent process) mentre il nuovo processo è detto **processo figlio** (child process). Il genitore ed il figlio hanno ciascuno un'immagine di memoria privata, di conseguenza se il genitore cambia qualcuna delle sue variabili, questa modifica non è visibile al figlio e viceversa.

I file aperti sono condivisi tra genitore e figlio, ovvero se un certo file era stato aperto dal

```

pid = fork(); /* se fork ha successo, nel genitore pid > 0 */
if (pid < 0) {
 gestione_errore(); /* la fork non ha successo (memoria o qualche tabella piena) */
} else if (pid>0) {
 /* il codice del genitore va qui */
} else{
 /* il codice del figlio va qui */
}

```

Figura 10.4 La creazione di un processo in UNIX.

genitore prima della `fork`, esso continuerà ad essere aperto per entrambi anche dopo. Le modifiche fatte al file da uno dei due saranno visibili anche all'altro, il che è ragionevole, perché le modifiche sono visibili anche a qualunque processo non correlato che apre il file.

Il fatto che le immagini di memoria, le variabili, i registri e qualsiasi altra cosa sono identiche per il genitore ed il figlio porta una piccola difficoltà: in che modo i processi possono sapere quale di essi dovrà girare con il codice del padre e quale con quello del figlio? Il segreto sta nel fatto che la chiamata di sistema `fork` restituisce uno 0 per il figlio ed un valore diverso da zero, detto `pid` (`process identifier` - identificatore di processo) per il genitore. Entrambi i processi di solito controllano il valore di ritorno ed agiscono di conseguenza, come mostrato in Figura 10.4.

I processi sono denominati con i loro PID; come si è visto, quando un processo è creato, il genitore fornisce il PID al figlio; se il figlio vuole conoscere il suo PID, c'è una chiamata di sistema, la `getpid`, che glielo restituisce. I PID vengono usati in più modi: per esempio, quando un figlio termina, al genitore è dato il PID del figlio che ha appena terminato. Ciò può essere importante perché un genitore può avere più figli e poiché i figli possono anch'essi avere altri figli, l'unico processo di origine può creare un intero albero di figli, nipoti e discendenti ulteriori.

In UNIX, i processi possono comunicare tra loro mediante lo scambio di messaggi. È possibile creare un canale tra due processi in cui un processo può scrivere un flusso di byte che l'altro processo può leggere. Questi canali sono chiamati pipe. La sincronizzazione è possibile perché quando un processo prova a leggere da un pipe vuoto rimane bloccato finché non sono disponibili i dati.

Le pipeline della shell sono implementate con pipe. Quando la shell vede un comando del tipo

```
sort <f | head
```

crea due processi, `sort` e `head`, e colloca tra questi un pipe in modo tale che lo standard output di `sort` sia connesso con lo standard input di `head`. In questo modo, tutti i dati che il `sort` scrive vanno direttamente a `head`, invece di andare su un file. Se il pipe si riempie, il sistema ferma l'esecuzione di `sort` finché `head` non ha rimosso un po' di dati dal pipe.

I processi possono comunicare anche in un altro modo: attraverso le interruzioni del software. Un processo può mandare un segnale (signal) ad un altro processo, inoltre essi possono comunicare al sistema cosa vogliono che accada quando arriva un segnale. Le scelte sono diverse: ignorarlo, prenderlo in considerazione o lasciare che esso elimini il

| Signal  | Causa                                                                   |
|---------|-------------------------------------------------------------------------|
| SIGABRT | Interrompe un processo e forza un dump della memoria                    |
| SICALRM | L'orologio di allarme è scattato                                        |
| SIGFPE  | È avvenuto un errore in virgola mobile (per esempio divisione per zero) |
| SIGHUP  | La linea telefonica che il processo stava usando è stata chiusa         |
| SIGILL  | L'utente ha usato la chiave DEL per interrompere il processo            |
| SIGQUIT | L'utente ha usato la chiave di richiesta di un dump di memoria          |
| SIGKILL | Segnale per eliminare un processo (non può essere bloccato o ignorato)  |
| SIGPIPE | Il processo ha scritto su un pipe che non ha lettori                    |
| SIGSEGV | Il processo ha fatto riferimento a un indirizzo di memoria non valido   |
| SIGTERM | Informa che un processo è terminato in modo normale                     |
| SIGUSR1 | Disponibile per scopi definiti dall'applicazione                        |
| SIGUSR2 | Disponibile per scopi definiti dall'applicazione                        |

Figura 10.5 I segnali richiesti da POSIX.

processo (per difetto). Se un processo decide di prendere in considerazione il segnale ad esso inviato, allora deve specificare la procedura di gestione del segnale; all'arrivo di un segnale, il controllo sarà passato immediatamente a tale gestore, e quando questi termina il suo lavoro, il controllo torna al processo chiamante, esattamente come succede per le interruzioni hardware di I/O. Un processo può solo mandare segnali ai membri del suo **gruppo di processi** (process group), formato dai genitori (e dagli antenati), dai fratelli, dai figli e dai discendenti. Il segnale a tutti i membri di un gruppo di processo può venir mandato con una sola chiamata di sistema.

I segnali sono usati anche per altri scopi. Per esempio, se un processo sta eseguendo un'operazione aritmetica in floating-point (virgola mobile), ed inavvertitamente divide un numero per zero, esso riceve un segnale SIGFPE (eccezione di floating-point). I segnali usati da POSIX sono elencati in Figura 10.5; molti sistemi UNIX hanno segnali aggiuntivi, ma i programmi che li usano non possono essere portati su altre versioni di UNIX.

### 10.3.2 Chiamate di sistema per la gestione dei processi in UNIX

Osserviamo adesso le chiamate di sistema UNIX per la gestione dei processi. Le principali sono elencate in Figura 10.6. La descrizione inizia con la chiamata **fork**, il solo modo per creare un nuovo processo. **fork** crea un duplicato esatto del processo originale, comprendente tutti i descrittori dei file, i registri e qualsiasi altra cosa. Dopo la **fork** il processo originale e la copia (il genitore ed il figlio) procedono il lavoro su strade separate. Tutte le variabili hanno valori identici al momento della **fork**, ma poiché l'immagine di memoria di tutto il genitore viene copiata per creare il figlio, le modifiche successive in uno di essi non hanno alcun effetto sull'altro. La chiamata **fork** restituisce un valore, che è zero nel figlio ed uguale all'identificatore del processo del figlio o **PID** nel genitore. Usando il **PID** restituito, i due processi possono distinguere tra genitore e figlio.

Nella maggior parte dei casi, dopo una **fork**, il figlio avrà bisogno di eseguire un codice diverso dal genitore. Si consideri il caso della shell: essa legge un comando dal terminale, chiede la **fork** di un processo figlio, attende che il figlio esegua il comando e legge il comando successivo quando il figlio ha terminato. Per attendere che il figlio finisca,

| Chiamata di sistema                             | Descrizione                                                   |
|-------------------------------------------------|---------------------------------------------------------------|
| pid = fork()                                    | Crea un processo figlio identico al padre                     |
| pid = waitpid(pid, &stato, opz)                 | Attende che un processo figlio abbia terminato                |
| s = execv(nome, argv, ambp)                     | Rimpiazza l'immagine di memoria di un processo                |
| exit(stato)                                     | Termina l'esecuzione e restituisce lo stato                   |
| s = sigaction(segnale, &azione, &vecchiaazione) | Specifica un'azione da intraprendere all'arrivo di un segnale |
| s = sigreturn(&context)                         | Ritorno dopo la gestione di un segnale                        |
| s = sigprocmask(come, &set, &vecchia)           | Esamina o cambia la maschera di segnali                       |
| s = sigpending(&set)                            | Recupera l'insieme di segnali bloccati                        |
| s = sigsuspend(nascherasegnale)                 | Sostituisce la maschera di segnali e sospende il processo     |
| s = kill(pid, segnale)                          | Manda un segnale ad un processo                               |
| residuo = alarm(secondi)                        | Definisce il clock di allarme                                 |
| s = pause()                                     | Sospende il chiamante fino al segnale successivo              |

Figura 10.6 Alcune chiamate di sistema legate ai processi. In caso di errore il codice restituito s è -1, pid è l'ID del processo e residuo è il tempo rimanente dall'ultimo allarme. I parametri delle funzioni sono esattamente quello che i loro nomi suggeriscono.

```

while (TRUE) { /* ripete per sempre */
 scrivi_prompt(); /* visualizza il prompt sul video */
 leggi_comando(comando, parametri); /* legge l'input dal terminale */

 pid = fork(); /* esegue una fork del processo figlio */
 if (pid < 0) {
 printf("Impossibile fork"); /* condizione di errore */
 continue; /* ripete il ciclo */
 }
 if (pid != 0) {
 waitpid (-1, &stato, 0); /* il genitore aspetta il figlio */
 } else {
 execve(comando, parametri, 0); /* il figlio esegue il lavoro */
 }
}

```

Figura 10.7 Una shell molto semplificata.

il genitore esegue una chiamata di sistema `waitpid`, che non fa altro che attendere che il figlio termini (uno qualsiasi dei figli nel caso ne esista più di uno). `waitpid` ha tre parametri: il primo permette al chiamante di attendere un determinato figlio (se è -1, sarà il primo figlio che termina); il secondo è l'indirizzo di una variabile che verrà aggiornata con lo stato di terminazione del figlio (terminazione normale o no, più il valore di uscita); il terzo determina se il chiamante si blocca, o se ritorna subito anche se nessun figlio è già terminato.

Nel caso della shell, il processo figlio deve eseguire il comando digitato dall'utente. Ciò viene realizzato tramite una chiamata di sistema `exec`, che ha l'effetto di sostituire l'intera immagine di memoria con il file nominato nel suo primo parametro. Nella Figura 10.7 si vede una shell molto semplificata che usa `fork`, `wait` ed `exec`.

Nel caso più generale, `exec` ha tre parametri: il nome del file che deve essere eseguito, un puntatore al vettore degli argomenti e un puntatore al vettore di ambiente. Tutti e tre questi parametri saranno descritti brevemente. Diverse procedure di libreria `exec1`, `execu`, `execle` ed `execve` vengono messe a disposizione per permettere di omettere i parametri o di specificarli diversamente. Tutte le procedure chiamano la chiamata di sistema basilare `exec`, e non esiste nessuna procedura di libreria di nome `exec`, bisogna usare una di esse.

Si consideri il comando

```
cp file1 file2
```

usato per copiare il `file1` sul `file2`. Dopo che la shell ha fatto una `fork`, il figlio localizza ed esegue il file `cp` e dà le informazioni sul file da copiare.

Il programma principale di `cp` (e di molti altri programmi) contiene la dichiarazione

```
main (argc, argv)
```

in cui `argc` è un contatore del numero di oggetti sulla linea di comando, compreso il nome del file; nell'esempio `argc` è 3.

Il secondo parametro, `argv`, è un puntatore ad un vettore. L'elemento *i* di questo vettore è un puntatore all'*i*-esima stringa nella linea di comando, nell'esempio `argv[0]` punterà alla stringa "cp", `argv[1]` punterà alla stringa di 5 caratteri "file1" e `argv[2]` a quella di 5 caratteri "file2".

Il terzo parametro di `main`, `envp`, è un puntatore all'ambiente, un vettore di stringhe contenente le assegnazioni della forma `nome = valore` usati per passare informazioni quali il tipo di terminale e il nome della directory di provenienza del programma. In Figura 10.7 non c'è nessun ambiente passato al figlio, per cui il terzo parametro di `execve` è uguale a zero.

Non bisogna spaventarsi se la chiamata `exec` sembra complicata, infatti è la più complessa delle chiamate di sistema. Le altre chiamate sono molto più semplici; come esempio di una chiamata semplice si consideri `exit`, che i processi useranno quando hanno terminato l'esecuzione. Essa ha un parametro, lo stato di uscita (tra 0 e 255), che è restituito al genitore nella variabile `stato` della chiamata di sistema `waitpid`. Il byte di ordine più basso della variabile `stato` contiene lo stato di terminazione, che normalmente è 0, ed altri valori indicanti le varie condizioni di errore. Il byte di ordine più alto contiene lo stato di uscita del figlio (tra 0 e 255). Per esempio, se il processo genitore esegue l'istruzione

```
n = waitpid (-1, &stato, 0);
```

esso sarà messo in attesa finché un processo figlio non termina. Si supponga che il figlio esca con 4 come parametro di `exit`, allora il genitore sarà attivato con `n` contenente il pid del figlio e `stato` a 0x0400 (in C, 0x è il prefisso per i numeri esadecimali). Il byte di ordine più basso dello `stato` è relativo al segnale; quello alto è il valore che il figlio restituisce alla chiamata dopo `exit`.

Se un processo termina e non ha il genitore in attesa, entra in uno stato di sospensione detto **zombie** (**zombie state**); quando il genitore si mette in attesa del figlio, quest'ultimo esce dallo stato zombie e termina la sua esecuzione.

Molte chiamate di sistema sono usate in diversi modi in relazione ai segnali. Per esempio, se un utente per sbaglio chiede a un editore di testi di visualizzare l'intero contenuto

to di un file molto lungo, e poi si accorge dell'errore, è necessario un modo per interrompere l'editore. La scelta normale per l'utente è quella di premere qualche tasto speciale (per esempio DEL o CTRL-C), che manda un segnale all'editore, che lo riceve ed interrompe la visualizzazione.

Per comunicare la sua disponibilità a ricevere questo segnale (o qualsiasi altro), il processo può usare la chiamata di sistema `sigaction`. Il primo parametro è il segnale da ricevere (vedere Figura 10.5), il secondo è un puntatore ad una struttura che contiene un puntatore alla procedura di gestione del segnale oltre che altri bit e flag vari. Il terzo parametro punta ad una struttura in cui il sistema restituisce l'informazione sulla gestione corrente del segnale, nel caso esso dovesse essere ripristinato in un secondo momento.

I driver dei segnali possono girare per tutto il tempo che ritengono necessario, benché, in pratica, essi di solito non impieghino molto tempo; quando la procedura di gestione del segnale termina, il processo ritorna al punto in cui era stato interrotto.

La chiamata di sistema `sigaction` può essere usata anche per fare in modo che un segnale sia ignorato, o per memorizzare di nuovo l'azione per difetto, che fa terminare forzatamente il processo.

L'uso del tasto DEL non è il solo modo per mandare un segnale: la chiamata di sistema `kill` permette di inviare un segnale ad un altro processo. La scelta del nome "kill" (uccidi) per questa chiamata di sistema non è delle migliori poiché più processi mandano segnali ad altri con l'intenzione di farsi ascoltare.

Per molte applicazioni in tempo reale, un processo ha bisogno di essere interrotto dopo un intervallo di tempo specifico per fare qualcosa, ad esempio per ritrasmettere un pacchetto potenzialmente perduto su una linea di comunicazione non affidabile. Per gestire questa situazione esiste la chiamata di sistema `alarm`, il cui parametro specifica un intervallo, in secondi, dopo il quale viene mandato al processo un segnale `SIGALARM`. Un processo può avere solo un segnale di `alarm` da evadere ad ogni istante. Se si fa una chiamata di `alarm` con un parametro di 10 secondi, e poi 3 secondi più tardi ne viene fatta un'altra con un parametro di 20 secondi, sarà generato un solo segnale 20 secondi dopo la seconda chiamata; il primo segnale è cancellato dalla seconda richiesta di `alarm`, e se il parametro di `alarm` è zero, qualsiasi segnale di `alarm` in attesa è cancellato. Se un segnale di `alarm` non è accettato, viene effettuata un'azione per difetto ed il processo segnalato viene fatto terminare forzatamente. Tecnicamente, i segnali di `alarm` si possono ignorare ma non ha senso farlo.

Qualche volta succede che un processo non ha niente da fare finché non arriva un segnale, per esempio, si consideri un computer su cui gira un programma di CAI (Computer Aided Instruction - insegnamento assistito dal calcolatore) che sta controllando la velocità di lettura e di apprendimento. Il programma visualizza una porzione di testo e poi chiama `alarm` per ricevere un segnale dopo 30 secondi. Mentre lo studente legge il testo, il programma non ha nulla da fare e potrebbe eseguire un lungo ciclo durante questa attesa, ma ciò significherebbe sprecare tempo di CPU che potrebbe essere utile ad un altro processo o ad un altro utente. Una soluzione migliore consiste nell'usare la chiamata di sistema `pause` che comunica ad UNIX di sospendere il processo fino all'arrivo del prossimo segnale.

## Le chiamate di sistema per la gestione dei thread

La prima versione di UNIX non possedeva thread, ed essi vennero aggiunti molti anni dopo: inizialmente, vi erano molti pacchetti di thread, ma la proliferazione di tali pacchetti rendeva difficoltoso il porting del codice. Per queste ragioni, le chiamate di sistema relative ai thread furono rese standard in POSIX (P1003.1c).

Nella specifica di POSIX non viene, però, definito in modo preciso se i thread devono essere implementati nel kernel o nello spazio utente. Il vantaggio di avere thread nello spazio utente è che questi possono essere implementati senza cambiare il kernel e che la commutazione di thread, in questo caso, risulta molto efficiente; lo svantaggio principale per i thread nello spazio utente è che se un thread si blocca (ad esempio, su un I/O, un semaforo o un fault di pagina), tutti gli altri thread nel processo a loro volta si bloccano, poiché il kernel riconosce solo un thread e non schedula il processo fino a quando il thread non si sblocca. Per queste ragioni, le chiamate definite nel P1003.1c sono state scelte attentamente in modo da poter essere implementate in entrambi i modi: fino a quando i programmi utente aderiscono alla semantica del P1003.1c, entrambe le implementazioni dovrebbero correttamente funzionare. Le chiamate a thread più comuni sono elencate in Figura 10.8; quando vengono utilizzati i thread in modo kernel, queste chiamate sono vere e proprie chiamate di sistema; se, invece, si usano chiamate a thread modo utente, esse sono implementate interamente in una libreria a tempo di esecuzione nello spazio utente.

Il lettore attento si sarà reso conto che adesso ci troviamo di fronte ad un problema tipografico: nel caso in cui sia il kernel ad amministrare i thread, le chiamate tipo "pthread\_create" sono chiamate di sistema e dovrebbero essere scritte così: `pthread_create`. Invece, se sono semplici chiamate di libreria nello spazio utente, la nostra convenzione di scrittura ci obbligherebbe ad usare il corsivo, come: `pthread_create`. Senza alcun tipo di pregiudizio, useremo il primo stile tipografico, anche nel prossimo capitolo, dove non è ben chiaro quali delle chiamate API Win32 siano vere chiamate di sistema. Ma non preoccupatevi, esistono cose peggiori: nel Report di Algol 68 vi era un carattere punto che cambiava leggermente la grammatica del linguaggio se stampato con il carattere sbagliato!

Esaminiamo, adesso, brevemente le chiamate a thread mostrate in Figura 10.8. La prima chiamata, `pthread_create`, crea un nuovo thread e viene chiamata da

```
err = pthread_create(&tid, attr, function, arg);
```

Questa chiamata crea, appunto, un nuovo thread nel processo in esecuzione, eseguendo la funzione `function` con `arg` come parametro. Il nuovo thread ID viene immagazzinato in memoria alla locazione puntata dal primo parametro; l'attributo `attr` può essere utilizzato per specificare gli attributi che il nuovo thread deve possedere, ad esempio la priorità

| Chiamata thread                    | Descrizione                                                     |
|------------------------------------|-----------------------------------------------------------------|
| <code>pthread_create</code>        | Crea un nuovo thread nello spazio degli indirizzi del chiamante |
| <code>pthread_exit</code>          | Termina il thread chiamante                                     |
| <code>pthread_join</code>          | Attende che un thread termini                                   |
| <code>pthread_mutex_init</code>    | Crea un nuovo mutex                                             |
| <code>pthread_mutex_destroy</code> | Distrugge un mutex                                              |
| <code>pthread_mutex_lock</code>    | Blocca un mutex                                                 |
| <code>pthread_mutex_unlock</code>  | Sblocca un mutex                                                |
| <code>pthread_cond_init</code>     | Crea una variabile di condizione                                |
| <code>pthread_cond_destroy</code>  | Distrugge una variabile di condizione                           |
| <code>pthread_cond_wait</code>     | Attende per una variabile di condizione                         |
| <code>pthread_cond_signal</code>   | Rilascia un thread aspettando una variabile di condizione       |

Figura 10.8 Le principali chiamate di thread di POSIX.

da assegnare in fase di scheduling. Se la procedura termina con successo, un thread in più risulta in esecuzione nello spazio degli indirizzi del chiamate rispetto a quanti erano in esecuzione prima della chiamata.

Un thread che ha compiuto il proprio lavoro e vuole terminare chiama `pthread_exit`; se, invece, un thread deve aspettare che un altro termini prima di concludersi chiama la funzione `pthread_join`: se il thread che stava aspettando è già terminato, `pthread_join` finisce immediatamente, altrimenti il thread viene messo in attesa.

I thread, inoltre, possono sincronizzarsi tra loro usando i mutex. Tipicamente un mutex sorveglia alcune risorse, come un buffer condiviso da due thread. Per essere certi che un solo thread per volta acceda alla risorsa condivisa, ogni thread deve bloccare un mutex al momento di utilizzo della risorsa e sbloccarlo non appena conclusa l'operazione. Se tutti i thread obbediscono a questo protocollo, le corse critiche possono essere evitate. I mutex sono come semafori binari, cioè semafori che possono prendere solo valori 0 e 1; il nome "mutex" deriva dal fatto che si usano per garantire la mutua esclusione sulle risorse.

Inoltre, i mutex possono essere creati e distrutti dalle chiamate `pthread_mutex_init` e `pthread_mutex_destroy`, rispettivamente; un mutex può trovarsi in uno dei seguenti due stati: bloccato o sbloccato (`locked` e `unlocked`). Quando un thread blocca un mutex sbloccato (usando `pthread_mutex_lock`), il blocco viene fissato ed il thread continua la propria esecuzione, se invece cerca di bloccare un mutex già bloccato da un altro thread, esso si blocca. Quando un thread che bloccava un mutex finisce di utilizzare la risorsa condivisa, sblocca il mutex chiamato la funzione `pthread_mutex_unlock`.

I mutex possono essere utilizzati per blocchi di risorse di breve durata, come nel caso della protezione di variabili condivise. Essi non possono essere utilizzati per sincronizzazioni a lungo termine, come il caso di dover aspettare un dispositivo di nastro momentaneamente occupato. Per la sincronizzazione a lungo termine, invece, sono disponibili le variabili di condizione (condition variable). Tali variabili vengono create e distrutte rispettivamente con le chiamate `pthread_cond_init` e `pthread_cond_destroy`.

Una variabile di condizione viene utilizzata quando un thread è in attesa, ed un altro thread la segnala. Per esempio, se un thread scopre che il dispositivo di nastro di cui necessita è occupato, può chiamare `pthread_cond_wait` sulla variabile di condizione relativa al dispositivo, ed usata da tutti per sincronizzarsi: quando il thread che usa il dispositivo a nastro ha finalmente concluso (magari anche ore dopo), esso usa `pthread_cond_signal` per liberare uno ed un solo thread tra quelli in attesa del dispositivo. Se non esistono thread in attesa del dispositivo, il segnale viene perso. In altre parole, le variabili di condizione non hanno contatori, a differenza dei semafori. Altre operazioni sono definite sui thread, mutex e sulle variabili di condizione.

### 10.3.3 Implementazione dei processi in UNIX

Un processo in UNIX è come un iceberg: ciò che si vede è solo la parte esterna ma c'è anche una parte importante che l'utente non vede. Ogni processo ha una parte utente che manda in esecuzione un programma utente; quando uno dei suoi thread opera una chiamata di sistema, esso passa in modo kernel, con una diversa mappa di memoria ed un completo accesso a tutte le risorse della macchina; in pratica è ancora lo stesso thread, ma possiede più potere ed un proprio stack e program counter in modo kernel. Essi sono importanti perché una chiamata di sistema può bloccare il processo a metà esecuzione, per esempio, nell'attesa che sia completata un'operazione su disco. Il contatore di programma ed i registri sono salvati, in modo che il thread possa ripartire in modo kernel.

Il kernel mantiene due strutture dati collegate ai processi, la **tabella dei processi** e la **struttura dell'utente**. La tabella dei processi è residente in memoria tutto il tempo e contiene l'informazione necessaria per tutti i processi, anche quelli che in quel momento non sono in memoria; la struttura utente viene ricoppiata o paginata su una zona disco riservata, quando il processo ad essa associato non è in memoria. Questa operazione viene eseguita per non occupare la memoria con informazioni che non sono necessarie in quel momento.

L'informazione contenuta nella tabella dei processi cade nelle seguenti categorie:

- Parametri di schedulazione.** La priorità del processo, il tempo di CPU usato più recentemente, il tempo di attesa più recente. Tutti questi parametri sono usati per decidere quale sarà il prossimo processo attivo.
- Immagine di memoria.** I puntatori al testo, ai dati, ai segmenti di stack o, se è usata la paginazione, alle tabelle delle pagine. Nel caso in cui il segmento di testo sia condiviso, il puntatore di testo punta alla tabella di testo condiviso. Quando il processo non è in memoria, in questa tabella si trova anche l'informazione relativa alla posizione delle sue parti su disco.
- Segnali.** Sono previste delle maschere che indicano quali segnali devono essere ignorati, quali presi in considerazione, quali devono essere temporaneamente bloccati e quali sono in fase di consegna.
- Varie.** Lo stato attuale del processo, l'evento che esso sta attendendo, se ve n'è qualcuno, il tempo del prossimo allarme, il PID, il PID del processo genitore, l'identificatore del gruppo e dell'utente.

La struttura utente contiene l'informazione che non è necessaria quando il processo non è eseguibile e non è in memoria centrale. Per esempio, sebbene per un processo sia possibile ricevere un segnale mentre è memorizzato su disco, esso non può leggere un file. Per questo motivo, le informazioni relative ai segnali devono essere nella tabella del processo, che è sempre in memoria centrale, mentre le informazioni relative ai descrittori del file si possono tenere nella struttura dell'utente e portare in memoria solo quando il processo è in fase di esecuzione.

La struttura utente comprende i seguenti oggetti:

- Registri di macchina.** Quando si genera un trap nel kernel, i registri macchina vengono salvati (compresi quelli in virgola mobile, se vengono usati).
- Stato di chiamata di sistema.** L'informazione relativa alla chiamata di sistema corrente, compresi i parametri ed i risultati.
- Tabella dei descrittori di file.** Quando viene richiesta una chiamata di sistema che coinvolge un descrittore di file, questo viene usato come indice in questa tabella per localizzare la struttura dati in memoria (l'i-node) che corrisponde al file.
- Addebito.** È un puntatore ad una tabella che mantiene traccia del tempo di CPU usato dal processo, in modo utente e kernel. Alcuni sistemi vi memorizzano anche i limiti sulla quantità di tempo di CPU che un processo può usare, la dimensione massima del suo stack, il numero di pagine che può consumare ed altre informazioni.
- Stack del kernel.** Uno stack prestabilito per l'uso da parte del kernel.

Tenendo presente queste tabelle, ora è facile spiegare come viene creato un processo in UNIX. Quando viene eseguita una chiamata di sistema `fork` il processo chiamante fa una trap al kernel e cerca un elemento libero nella tabella dei processi da usare per il

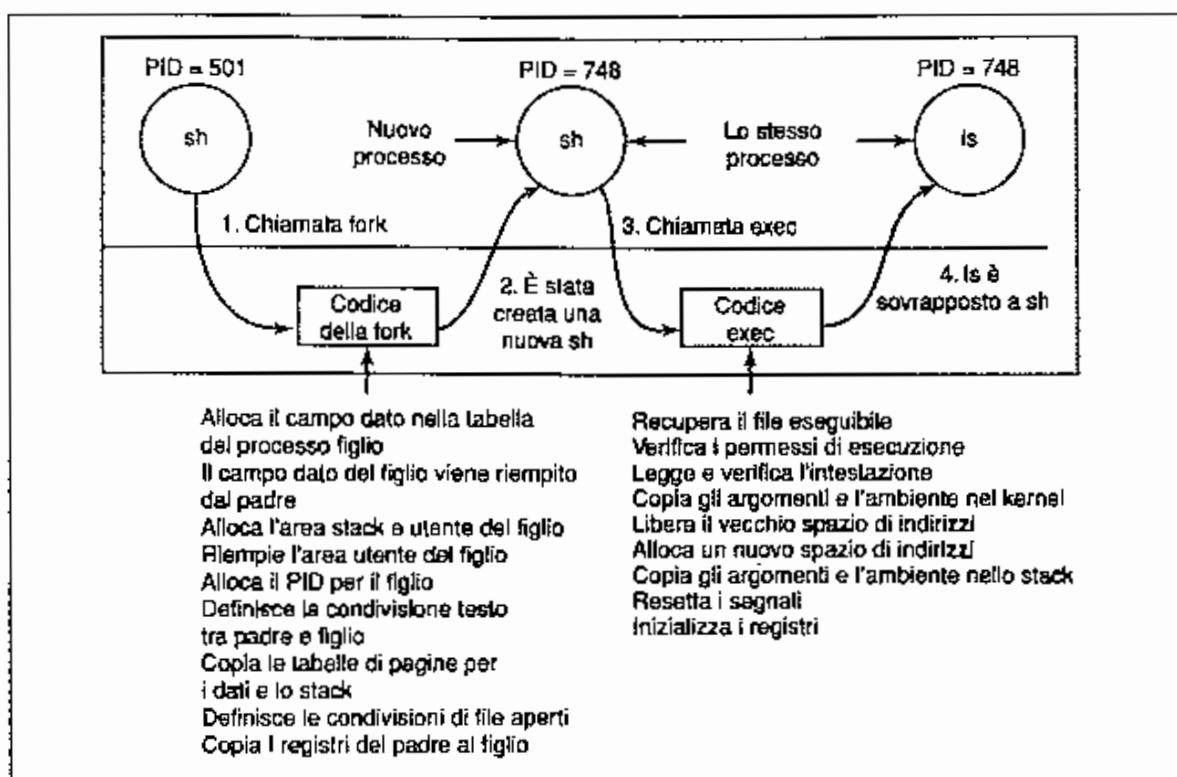


Figura 10.9 I passi per l'esecuzione del comando `ls` scritto da shell.

figlio. Se ne trova uno, copia tutte le informazioni dall'elemento della tabella del processo del genitore sull'elemento del figlio. Successivamente alloca la memoria per i dati del figlio ed i segmenti di stack e vi fa una copia esatta dei segmenti del genitore. La struttura utente è mantenuta adiacente al segmento di stack ed è copiata con esso; il segmento testo può essere copiato o condiviso, essendo a sola lettura. Ora il figlio è pronto per andare in esecuzione.

Quando un utente digita un comando, ad esempio `ls` sul terminale, la shell crea un nuovo processo producendo con una `fork` un clone di se stessa. La nuova shell, quindi, chiama `exec` per sovrascrivere con il contenuto del file eseguibile `ls` la propria memoria. I passi appena descritti sono elencati in Figura 10.9.

Da qui, il meccanismo per la creazione di un nuovo processo, è abbastanza immediato: viene creato un nuovo campo dato nella tabella dei processi ed una nuova area utente per il processo figlio, riempiendoli in prevalenza con le informazioni del genitore. A tale figlio viene assegnato un PID, viene definita la sua mappa di memoria, e gli viene dato accesso per la condivisione dei file del padre. Infine, vengono impostati i registri in modo che sia pronto per essere mandato in esecuzione.

Come principio, occorrerebbe fare una copia completa dello spazio degli indirizzi del processo padre, visto che il comando `fork` dice che non è possibile avere memoria condivisa tra processo padre e figlio. Ad ogni modo, fare la copia della memoria è abbastanza costoso e quindi i moderni sistemi UNIX lo evitano: danno al processo figlio le tabelline di pagine, che puntano alle pagine del processo padre, marcate solo in lettura. Se il figlio cerca di scrivere su tali pagine, genera una trap di protezione, di cui il kernel si accorge e, di conseguenza, alloca una nuova copia della pagina per il figlio e la marca come lettura/scrittura. In questo modo, solo le pagine che devono essere effettivamente scritte vengono copiate. Questo meccanismo viene chiamato `copy-on-write` (copia-su-

scrittura), e permette che non si debbano avere necessariamente due copie del programma in memoria, risparmiando memoria RAM.

Dopo che il processo figlio inizia la propria esecuzione, il codice (una copia della shell) esegue una nuova `exec`, con il nome del comando come parametro. Quindi, il kernel trova e verifica il file eseguibile, copia gli argomenti e le stringhe di ambiente e rilascia il vecchio spazio dei processi e le relative tabelle di pagine.

Solo in questo momento occorre creare un nuovo spazio degli indirizzi da riempire: se il sistema supporta la mappatura dei file, come fanno System V, BSD e la maggior parte dei sistemi UNIX, la nuova tabella delle pagine viene inizializzata per indicare che non vi sono pagine in memoria, fatta eccezione, forse, per una pagina stack, ma lo spazio degli indirizzi è memorizzato sul file eseguibile su disco. Quando il nuovo processo è in esecuzione, ottiene immediatamente un fault di pagina che permette di caricare la prima pagina di codice dal file eseguibile. In questo modo, nulla deve essere caricato in anticipo, in modo che i programmi possano iniziare velocemente e richiedere le pagine che effettivamente servono e nulla di più. Infine, gli argomenti e le stringhe di ambiente vengono copiate nel nuovo stack, i segnali vengono azzerati ed i registri vengono inizializzati a zero. A questo punto, un nuovo comando può iniziare la propria esecuzione.

## I thread in UNIX

L'implementazione dei thread dipende dal fatto che essi siano o meno supportati dal kernel. Se non lo sono, come nel caso di 4BSD, l'implementazione è interamente in una libreria dello spazio utente; se, invece sono supportati dal kernel, come nel caso di System V e Solaris, il kernel ha del lavoro da fare. Abbiamo discusso in generale dei thread nel Capitolo 2: qui cercheremo di fare alcune precisazioni circa i thread ed il kernel in UNIX.

Il problema principale dell'introduzione dei thread è il mantenimento delle semantiche tradizionali corrette di UNIX. Si consideri per iniziare la `fork`: si supponga che un processo con thread multipli in modo kernel operi una chiamata `fork`. Si dovrebbero creare anche tutti gli altri thread nel nuovo processo? Per adesso si supponga che la risposta sia sì; se però uno degli altri thread era bloccato leggendo da tastiera, il corrispondente thread nel nuovo processo dovrebbe pure esso bloccarsi sulla lettura da tastiera? Se sì, quale dei due thread recupererà la successiva riga digitata? Se no, cosa dovrebbe fare il thread corrispondente nel nuovo processo? Lo stesso tipo di problematiche si possono riscontrare in molte altre operazioni che possono essere fatte dai thread. In un processo a thread unico il problema non esiste, visto che l'unico thread non può essere bloccato quando chiama una `fork`. Si consideri adesso il caso in cui gli altri thread del processo padre non vengono creati nel processo figlio, e che uno dei thread non creati tenga un mutex, che l'unico thread nel nuovo processo cerca di acquisire dopo aver fatto la `fork`. In questo caso il mutex non verrà mai rilasciato ed il thread del processo figlio aspetterà all'infinito. Quello che è evidente è che non esiste una soluzione semplice ai numerosi problemi.

L'I/O dei file è un altro problema: si supponga che un thread sia bloccato durante la lettura da un file, e che un altro thread chiuda il file o operi un `lseek` per cambiare il puntatore corrente al file. Cosa accade dopo? Chissà.

La gestione dei segnali è un altro punto interrogativo, i segnali, infatti, dovrebbero essere diretti ad un thread specifico o al processo in generale? Probabilmente il thread che causa una SIGFPE (floating point exception, eccezione di aritmetica in virgola mobile) dovrebbe poterla catturare. Cosa succede se non la recupera? Dovrebbe essere ucciso quel solo thread o tutti i thread del processo? Si consideri poi il segnale SIGINT generato dall'utente alla tastiera. Quale thread dovrebbe occuparsi del recupero di tale segnale? Un insieme comune di maschere di segnali dovrebbe essere condiviso da tutti i thread? Le

soluzioni a tutti questi problemi non sono così semplici da trovare, e cercare di tenere corretta la semantica dei thread è un affare decisamente non banale, per non parlare di come scriverne il codice.

## Thread in Linux

Linux supporta i thread in un modo particolare, che vale la pena di conoscere; l'implementazione è basata sulle idee del 4.4 BSD. Ma i thread di kernel non erano stati abilitati in tale distribuzione poiché Berkeley finì i soldi prima che la libreria C potesse essere sistemata per risolvere i problemi discussi sopra.

Una nuova chiamata di sistema, `clone`, è il cuore dell'implementazione dei thread in Linux, e non è presente in nessun'altra versione di UNIX. Essa viene chiamata come segue:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

La chiamata crea un nuovo thread, o nel processo corrente, o in un nuovo processo, in base al valore dell'argomento `sharing_flags`. Se il nuovo thread viene creato nel processo corrente, esso condivide lo spazio degli indirizzi con i thread esistenti, ed ogni successiva scrittura di un qualsiasi byte nello spazio degli indirizzi da un qualsiasi thread diventa immediatamente visibile a tutti gli altri thread del processo. D'altra parte, se lo spazio degli indirizzi non è condiviso, il nuovo thread ottiene una copia esatta dello spazio degli indirizzi, e le successive scritture del thread non risultano visibili ai vecchi thread. Questa semantica è la stessa della `fork`.

In entrambi i casi, il nuovo thread incomincia l'esecuzione della `function` con parametro `arg`. Inoltre il nuovo thread ottiene il proprio stack privato con il puntatore dello stack inizializzato a `stack_ptr`.

Il parametro `sharing_flags` non è altro che una mappa di bit che permette una condivisione a grana più fine rispetto ai sistemi UNIX tradizionali: come mostrato in Figura 10.10 vengono definiti 5 bit ed ogni bit controlla un certo aspetto della condivisione e può essere definito in modo indipendente dagli altri. Il bit `CLONE_VM` determina se la memoria virtuale (cioè lo spazio degli indirizzi) è condivisa con i vecchi thread o copiata. Se il bit è posto a 1, il nuovo thread viene aggiunto fra quelli esistenti e quindi la chiamata `clone` di fatto crea un nuovo thread nel processo esistente. Al contrario se il bit è posto a 0, il nuovo thread ottiene il proprio spazio degli indirizzi; in questo modo le relative istruzioni `STORE` non risultano visibili ai thread esistenti. Questo comportamento è simile alla chiamata `fork` fatta eccezione per le osservazioni che seguono: la creazione di un nuovo spazio degli indirizzi è effettivamente la definizione di un nuovo processo.

| Flag                       | Se settato significa...                       | Se non settato significa...           |
|----------------------------|-----------------------------------------------|---------------------------------------|
| <code>CLONE_VM</code>      | Crea un nuovo thread                          | Crea un nuovo processo                |
| <code>CLONE_FS</code>      | Condivide umask, radice e directory di lavoro | Non le condivide                      |
| <code>CLONE_FILES</code>   | Condivide i descrittori di file               | Copia i descrittori di file           |
| <code>CLONE_SIGHAND</code> | Condivide la tabella di gestori di segnale    | Copia la tabella                      |
| <code>CLONE_PID</code>     | Il nuovo thread recupera il vecchio PID       | Il nuovo thread prende il proprio PID |

Figura 10.10 I bit nella mappa di bit `sharing_flags`

Il bit `CLONE_FS` controlla la condivisione della directory di radice e di lavoro ed il flag `umask`; se vale 1 i vecchi ed i nuovi thread condividono le directory di lavoro anche se il nuovo thread possiede il proprio spazio degli indirizzi. Questo vuol dire che una chiamata `chdir` da parte di un thread cambia la directory di lavoro anche degli altri thread, anche se questi posseggono i propri spazi di indirizzi, mentre in UNIX, una chiamata a `chdir` da parte di un thread cambia la directory di lavoro agli altri thread nello stesso processo, ma mai per thread in altri processi. Per questa ragione, questo bit rende possibile un tipo di condivisione che in Unix non è ottenibile.

Il bit `CLONE_FILES` è analogo al bit `CLONE_FS`: se vale 1, il nuovo thread condivide i propri descrittori di file con i vecchi thread cosicché chiamate di `lseek` da parte di un thread sono visibili da altri come normalmente avviene tra thread nello stesso processo ma non tra thread di processi diversi. Similmente il bit `CLONE_SIGHAND` attiva o disattiva la condivisione della tabella dei gestori di segnali; se tale tabella è condivisa anche tra thread in diversi spazi di indirizzi, il cambiamento di un gestore in un thread influenza i gestori negli altri thread. Infine `CLONE_PID` controlla se il nuovo thread ottiene il proprio PID o condivide il PID del genitore: questa caratteristica è necessaria in fase di avvio del sistema; e i processi utente non possono attivare questo bit.

Questa condivisione più puntuale è possibile poiché Linux mantiene strutture dati separate per i vari oggetti elencati all'inizio della Sezione 10.3.3 (parametri di scheduling, immagini di memoria, eccetera). La tabella dei processi e la struttura utente puntano semplicemente a queste strutture dati, cosicché è semplice creare un nuovo campo dato nella tabella dei processi per ogni thread clonato, e far puntare tale campo dato alle strutture dati, alla memoria e allo scheduling del vecchio thread o a copie di queste. Ad ogni modo, tale condivisione puntuale non è sempre utile: non essendo supportata da UNIX, un programma Linux che la utilizza non risulta portabile su ambiente UNIX.

## Scheduling in UNIX

Si esamini ora l'algoritmo di schedulazione di UNIX. Poiché UNIX è un sistema in multiprogrammazione, il suo algoritmo di schedulazione è stato progettato per fornire dei buoni tempi di risposta ai processi interattivi. Esso è un algoritmo a due livelli: il livello più basso sceglie il prossimo processo da eseguire dall'insieme dei processi in memoria pronti per l'esecuzione, ed il livello superiore muove i processi tra la memoria e il disco in modo che tutti i processi abbiano la possibilità di risiedere in memoria e di essere eseguiti. Ogni versione di UNIX ha lo scheduler a basso livello leggermente diverso, ma i più seguono lo schema generale che descriveremo.

Il livello più basso dell'algoritmo usa diverse code, ad ognuna di esse è associato un insieme (disgiunto) di valori di priorità. I processi eseguiti in modalità utente (la parte superiore dell'iceberg) hanno valori positivi; i processi eseguiti in modalità kernel (che eseguono le chiamate di sistema) hanno valori negativi. I valori negativi hanno la priorità maggiore e i valori positivi hanno la priorità più bassa, come mostrato in Figura 10.11. Nelle code sono allocati solo i processi che sono in memoria pronti a girare, poiché la scelta deve essere fatta tra di essi.

Quando gira lo scheduler (livello basso), esso cerca la coda con priorità più alta (valore negativo, massimo valore assoluto) finché non ne trova una che è occupata. Viene scelto ed eseguito il primo processo che è su quella coda, e tale esecuzione dura un quanto di tempo, di solito 100 ms, o finché esso non si blocca. Se un processo esaurisce il suo quanto, è messo in fondo alla coda e l'algoritmo di schedulazione ricomincia a girare. I processi con la stessa priorità condividono la CPU usando l'algoritmo round robin.

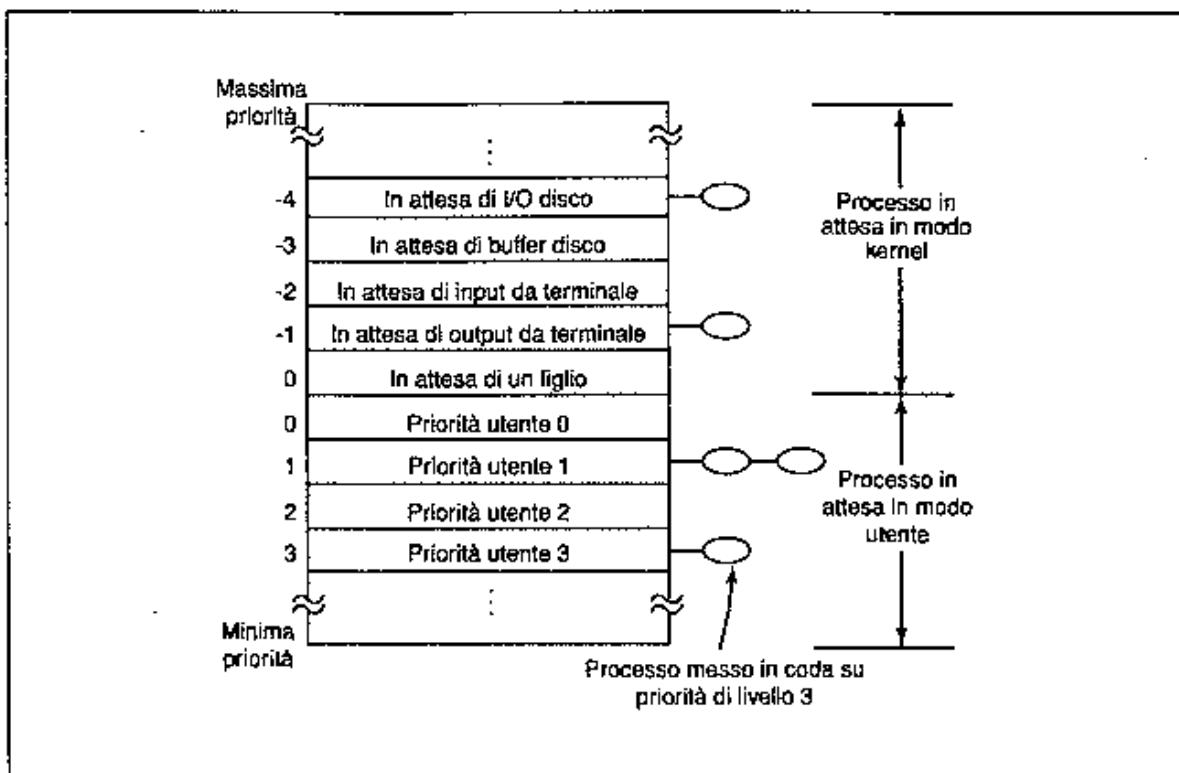


Figura 10.11 Lo scheduler UNIX è basato su una struttura a code multilivello.

Una volta ogni secondo tutte le priorità dei processi sono ricalcolate secondo la formula

$$\text{priorità} = \text{base} + \text{uso\_della\_CPU} * \text{nice}$$

Ogni processo viene poi attaccato ad una coda di Figura 10.11 in base alla nuova priorità, il numero di tale coda si ottiene dividendo la priorità per una costante. Proviamo adesso ad esaminare brevemente ognuna delle tre componenti presenti nella formula appena introdotta.

*uso\_della\_CPU* rappresenta il numero medio di tic dell'orologio per ogni secondo che il processo ha utilizzato durante gli ultimi secondi precedenti; il contatore dell'uso della CPU, presente come campo dato nella tabella del processo in esecuzione, viene aumentato di 1 ogni volta che l'orologio batte un colpo. Tale contatore sarà aggiunto alla priorità del processo, ottenendo un valore più alto e posizionando quindi il processo stesso in una coda di priorità più bassa.

Ad ogni modo, UNIX non è un sistema che punisce i processi per sempre, e per questo *uso\_della\_CPU* decade nel tempo: in particolare diverse versioni di UNIX operano il decadimento in modo leggermente diverso. Uno dei modi utilizzati è stato quello di aggiungere il valore corrente di *uso\_della\_CPU* al numero di tic acquisiti nell'ultimo intervallo di tempo,  $\Delta T$ , e dividere il risultato per 2: questo algoritmo pesa i più recenti intervalli di tempo con 1/2, quelli precedenti di 1/4, e così via dicendo. In genere possono essere utilizzati altri algoritmi, con diverse politiche di pesatura dei  $\Delta T$ , ma quello appena descritto risulta un algoritmo molto veloce poiché opera solamente una addizione ed uno shift.

Ogni processo ha, inoltre, un valore *nice* associato. Il valore di base è 0, e l'intervallo di valori possibile è -20, +20; quindi con una chiamata di sistema *nice*, il processo può dare al proprio *nice* un valore tra 0 e 20. Un utente che sta calcolando il valore di  $n$  con

un miliardo di cifre potrebbe mettere questa chiamata nel suo programma per essere gentile con gli altri utenti; solo l'amministratore di sistema può passare i valori tra -20 e -1, ottenendo quindi un *miglior servizio* ai propri processi. Dedurre il perché di questa regola viene lasciato come esercizio al lettore.

Quando un processo fa una trap al kernel per una chiamata di sistema, è possibile che il processo si debba bloccare, prima di completare la chiamata di sistema e ritornare in modalità utente. Per esempio, esso può fare una chiamata di sistema `waitpid` e deve attendere che uno dei suoi figli termini; ma può anche dover attendere un input da terminale o il completamento di un I/O su disco, e queste sono solo due delle molteplici possibilità che si possono incontrare. Al momento del suo blocco viene rimosso dalla struttura di coda poiché non è abilitato a girare.

Comunque, quando si ha l'evento di cui il processo era in attesa, esso viene inserito in una coda con un valore negativo. La scelta della coda avviene in base al tipo di evento di cui era in attesa; nella Figura 10.11, la più alta priorità è data all'I/O da disco, per cui un processo che ha appena letto o scritto un blocco dal disco probabilmente riceverà l'uso della CPU entro 100 ms. La priorità relativa dell'I/O da disco, o da terminale, eccetera, è cablata nel sistema operativo e può essere modificata solo cambiando alcune costanti nel codice sorgente e ricompilando il sistema. Questi valori (negativi) sono rappresentati dal campo `base` nella formula data prima e sono abbastanza diversi l'uno dall'altro in modo che processi che sono stati riavviati per varie ragioni finiscono ben divisi, in code differenti.

L'idea di base di questo schema è quella di portare velocemente i processi fuori dal kernel. Se un processo sta provando a leggere un file dal disco, facendolo attendere un secondo tra le chiamate di sistema `read` lo si rallenterà in maniera considerevole; è molto meglio farlo girare subito dopo che è stata completata ogni richiesta, così esso può fare la prossima richiesta velocemente. Allo stesso modo, se un processo era bloccato in attesa di un input da terminale, è certamente un processo interattivo a cui si dovrebbe dare una priorità alta, in modo che appena è pronto possa fornire un buon servizio all'utente. Da questo punto di vista, i processi CPU bound (cioè quelli sulle code di valori positivi) in pratica sono serviti solo quando tutti i processi I/O bound e quelli interattivi sono bloccati.

## Scheduling in Linux

L'argomento scheduling è uno di quelli in cui UNIX e Linux sono leggermente diversi. Fino ad ora abbiamo esaminato l'algoritmo di scheduling di UNIX, adesso osserveremo quello di Linux. Tanto per incominciare, i thread Linux sono solamente thread kernel e, per questo, lo scheduling è sempre basato sui thread e mai sui processi. In particolare Linux distingue tre tipologie di thread nel contesto scheduling:

1. FIFO in tempo reale (real-time).
2. Round robin in tempo reale.
3. Timesharing.

I thread del primo gruppo sono con priorità più alta e non sono prerilasciabili se non da nuovi thread dello stesso gruppo. I thread round robin in tempo reale, invece, sono prerilasciabili dal clock del sistema: se più thread round robin in tempo reale sono pronti nello stesso istante, ognuno viene eseguito per un quanto, e quindi posto alla fine della lista che contiene tutti i thread round robin in tempo reale da eseguire. Nessuna di queste due tipologie di thread è veramente in tempo reale: non possono essere definite scadenze e

non vengono date garanzie; queste classi sono semplicemente classi a priorità più alta rispetto all'ultima. La ragione per la quale vengono chiamate in tempo reale è che Linux è conforme allo standard P1003.4 (estensioni "real-time" di UNIX) che usa questa definizione.

Ogni thread ha una priorità di scheduling il cui valore per difetto è 20, ma che può variare usando la chiamata di sistema `nice(value)` ottenendo il nuovo valore dato da `20-value`. Poiché `value` deve essere compreso tra -20 e +19 le priorità cadono sempre nell'intervallo [1,40]. L'idea di base di questa politica è che la qualità del servizio ad un thread deve essere proporzionale alla priorità, in modo che i thread con priorità più alta possano essere eseguiti più velocemente dei thread a priorità bassa.

In aggiunta alla priorità, ogni thread possiede un quanto associato che stabilisce il numero di tic del clock che il thread può utilizzare in fase di esecuzione. Il clock gira a 100Hz per difetto e quindi ogni tic è di 10 ms, detto anche jiffy. L'algoritmo di scheduling utilizza il quanto e la priorità associati ai vari thread come segue: in primo luogo viene calcolata la bontà (goodness) di ogni thread pronto, applicando le seguenti regole

```
if (class == real_time) goodness = 1000 + priorità;
if (class == timesharing && quanto > 0) goodness = quanto + priorità;
if (class == timesharing && quanto == 0) goodness = 0;
```

Entrambe le classi di thread in tempo reale cadono nella prima regola; marcando un thread come in tempo reale si è sicuri che ottiene una priorità più alta dei thread di tipo timesharing. Inoltre, l'algoritmo possiede una ulteriore caratteristica: se il processo che gira per ultimo lascia qualche quanto, ottiene un punto di bonus per vincere in successive contese. L'idea, qui, è che a parità di tutto il resto, è più efficiente eseguire il processo precedente poiché la pagine ed i blocchi di cache non devono essere caricati.

Detto questo, l'algoritmo di scheduling è molto semplice: quando viene fatta una decisione di schedulare un thread, viene selezionato il thread con goodness più elevata. Ad ogni tic del clock il relativo quanto viene decrementato di 1, e la CPU gli viene sottratta se si verifica uno dei seguenti casi:

1. Il quanto del thread è 0.
2. Il thread viene bloccato su un I/O, un semaforo o qualcosa' altro.
3. Un thread precedentemente bloccato ma con goodness migliore diventa pronto.

Poiché i quanti continuano ad essere decrementati, prima o poi, tutti i thread pronti avranno consumato il loro quanto, e si troveranno con il quanto a 0, mentre i thread I/O bound bloccati potrebbero avere ancora valori di quanto diversi da zero. A questo punto, l'algoritmo ricalcola il quanto di tutti i thread, pronti o bloccati, mediante la regola:

$$\text{quanto} = (\text{quanto}/2) + \text{priorità}$$

dove il nuovo quanto è calcolato in jiffy. Un thread compute-bound avrà il quanto a zero, ed otterrà come nuovo quanto il valore della priorità relativa; ma un thread I/O-bound, invece, avrà ancora il quanto diverso da zero ed otterrà un valore nuovo maggiore. Se non viene utilizzata la chiamata di sistema `nice`, la priorità sarà 20 e quindi il quanto, per il primo thread, assume valore di 20 jiffy, ovvero 200ms. Per thread I/O bound, il quanto potrebbe valere 20 al momento del ricalcolo, cosicché il nuovo quanto diventerebbe  $20/2 + 20 = 30$  jiffy. Se dovesse avvenire una nuova operazione di ricalcolo del quanto prima che il thread abbia utilizzato un tic del clock, il suo nuovo quanto sarebbe

$30/2 + 20 = 35$  jiffy. Il valore asintotico in jiffy è il doppio della priorità. Di conseguenza, i thread I/O bound ottengono quanti sempre più grandi e quindi una goodness maggiore, e così vengono preferiti in fase di scheduling ai thread CPU bound.

Infine, quando i thread compute-bound si stanno contendendo la CPU, quelli con priorità più alta la ottengono per una maggiore frazione di tempo. Si consideri, ad esempio, il caso di due thread *A* e *B*, il primo con priorità 20 ed il secondo con priorità 5: *A* ha la precedenza e 20 tic dopo ha concluso il proprio quanto; in seguito *B* utilizza il proprio quanto per 5 tic. A questo punto sono ricalcolati i quanti: *A* ottiene 20 e *B* nuovamente 5; questo per sempre, concedendo ad *A* l'80% della CPU mentre a *B* solo il 20%.

### 10.3.4 Avviare UNIX

L'esatto modo in cui il sistema UNIX viene avviato varia da sistema a sistema: qui di seguito vedremo brevemente come 4.4BSD viene avviato, ma le idee sono simili anche per le altre versioni. Quando un computer viene acceso, viene letto in memoria ed eseguito il primo settore del disco di avvio (il record di avvio principale), che contiene un piccolo programma (512-byte) che carica, dal dispositivo di avvio (di solito un disco IDE o SCSI), un programma standalone chiamato *boot*. Questo, in primo luogo, copia se stesso in un certo indirizzo alto di memoria, e libera la memoria più in basso per il sistema operativo.

Una volta posizionato al nuovo indirizzo di memoria, *boot* legge la directory di radice del dispositivo di avvio, per farlo deve capire il formato del file system e delle directory. Successivamente legge nel kernel del sistema operativo e vi salta dentro. A questo punto, *boot* ha concluso il proprio lavoro ed il kernel inizia la propria esecuzione.

Il programma di inizializzazione del kernel è in codice assembler, che dipende fortemente dalla macchina su cui sta girando. Tipicamente il kernel inizializza il proprio stack, identifica il tipo di CPU e calcola il quantitativo di RAM a disposizione, disabilita le interruzioni, abilita l'MMU ed, infine, chiama la procedura in C *main* perché avvii la parte principale del sistema operativo.

Il codice C deve inizializzare un certo numero di cose, ma più logiche che fisiche ed inizia allocando un buffer di messaggi per la diagnostica di eventuali problemi; nel frattempo, a schermo vengono scritti messaggi su cosa sta accedendo, in modo da poterli comunicare, in caso di errore, ad un programma di diagnostica. Si pensi a questa operazione come alla scatola nera di un aeroplano che memorizza tutto quello che sta accadendo al sistema.

Come passo successivo, vengono allocate le strutture dati del kernel, la maggior parte a dimensione fissa, ma poche (come la cache di buffer e certe strutture per le tabelle di pagine) a dimensione che varia in base alla RAM disponibile.

A questo punto il sistema inizia l'autoconfigurazione utilizzando i file di configurazione, che comunicano quali dispositivi di I/O devono essere presenti, e controlla quali effettivamente lo sono. Se un dispositivo risponde correttamente viene aggiunto nella tabella dei dispositivi legati al computer; se non risponde, viene considerato assente e ignorato fino all'avvio successivo.

Dopo aver determinato la lista dei dispositivi presenti, vengono caricati i driver associati: è questo il punto in cui i sistemi UNIX differiscono tra loro; in particolare, 4.4BSD non può caricare i driver in modo dinamico, e se un driver di dispositivo I/O non è stato collegato staticamente, tale dispositivo non può essere utilizzato. Di contro, altre versioni di UNIX, come Linux, possono caricare dinamicamente i driver (come possono fare tutte le versioni di Windows e di MS-DOS, fra l'altro).

Le argomentazioni pro e contro il caricamento dinamico dei driver sono degne di nota. La motivazione principale a favore del caricamento dinamico è che un singolo binario può essere venduto a diversi clienti, con configurazioni diverse, caricando dinamicamente i driver necessari alle varie configurazioni, magari dalla rete. Lo svantaggio principale è la mancanza di sicurezza in fase di caricamento dinamico. Se si deve gestire un sito sicuro, come il database di una banca o un server web di una società, probabilmente si desidera che nessuno possa inserire del codice nel kernel del sistema. L'amministratore di sistema può mantenere i sorgenti ed i file oggetto su una macchina sicura, su cui costruire il sistema, e passare il binario solo a chi di dovere. Se i driver non possono essere caricati dinamicamente, non è possibile che qualcuno che conosce la password da superuser, magari un operatore, manometta il codice del sistema. In più, in tali grandi siti la configurazione hardware è conosciuta a priori, al tempo di collegamento, ed i cambiamenti sono abbastanza rari, tanto da permettere il ricollegamento del sistema quando necessario.

Una volta configurato l'intero hardware, il processo 0 deve essere preparato, costruendo la sua stack, ed avviato. Esso è responsabile dell'inizializzazione del sistema, programma il clock in tempo reale, monta il file system radice, e crea *init*, il processo 1, ed il demone di pagina (il processo 2).

*init* controlla i propri flag per vedere se il sistema è multiutente o a utente singolo. Nel secondo caso, *init* fa una *fork*, ed il figlio lancia (*exec*) una shell, ed attende che il processo esca. Nel caso di multiutenza, invece, *init* fa la *fork*, ed il processo figlio (shell) esegue uno script, */etc/rc*, che effettua un controllo sulla consistenza del file system, monta file system addizionali, e così via. Questo processo poi legge un file, */etc/ttys*, che dice quanti terminali ha il sistema e fornisce determinate informazioni su ognuno di essi. In un secondo momento, *init* crea un processo figlio per ogni terminale e si pone in attesa finché uno dei figli non termina.

Ogni figlio esegue il programma *getty*, che imposta la velocità della linea del relativo terminale (ad esempio, può essere un *modem*), e quindi scrive sul video

*login:*

e prova a leggere il nome dell'utente dalla tastiera. Quando qualcuno fornisce un nome di login da terminale, *getty* termina eseguendo */bin/login*, il programma che chiede la password, la critta e la confronta con la password crittata memorizzata nel file delle password, */etc/passwd*. Se è corretta, *login* carica al suo posto la shell dell'utente, che attende il primo comando, nel caso contrario *login* chiede un altro nome utente. Questo meccanismo è mostrato in Figura 10.12 per un sistema con tre terminali.

In figura, il processo *getty* che gira per il terminale 0 è ancora in attesa dell'input, quello del terminale 1 ha letto il nome utente, quindi *getty* è stato sovrascritto da *login*, che è in attesa di password. Intanto, sul terminale 2 si è avuto un login corretto, qui la shell ha scritto il prompt (%), l'utente ha digitato

*cp f1 f2*

ed il programma corrispondente è stato lanciato da una *fork* della shell, seguita da una *exec* di *cp*, che quindi sta girando come figlio della shell. La shell è bloccata in attesa che il figlio termini, e a quel punto la shell manderà un prompt e leggerà dalla tastiera. Se l'utente del terminale 2 avesse digitato *cc* invece di *cp*, sarebbe stato attivato il programma principale del compilatore C che avrebbe creato più processi, uno alla volta, per eseguire i vari passi di compilazione.

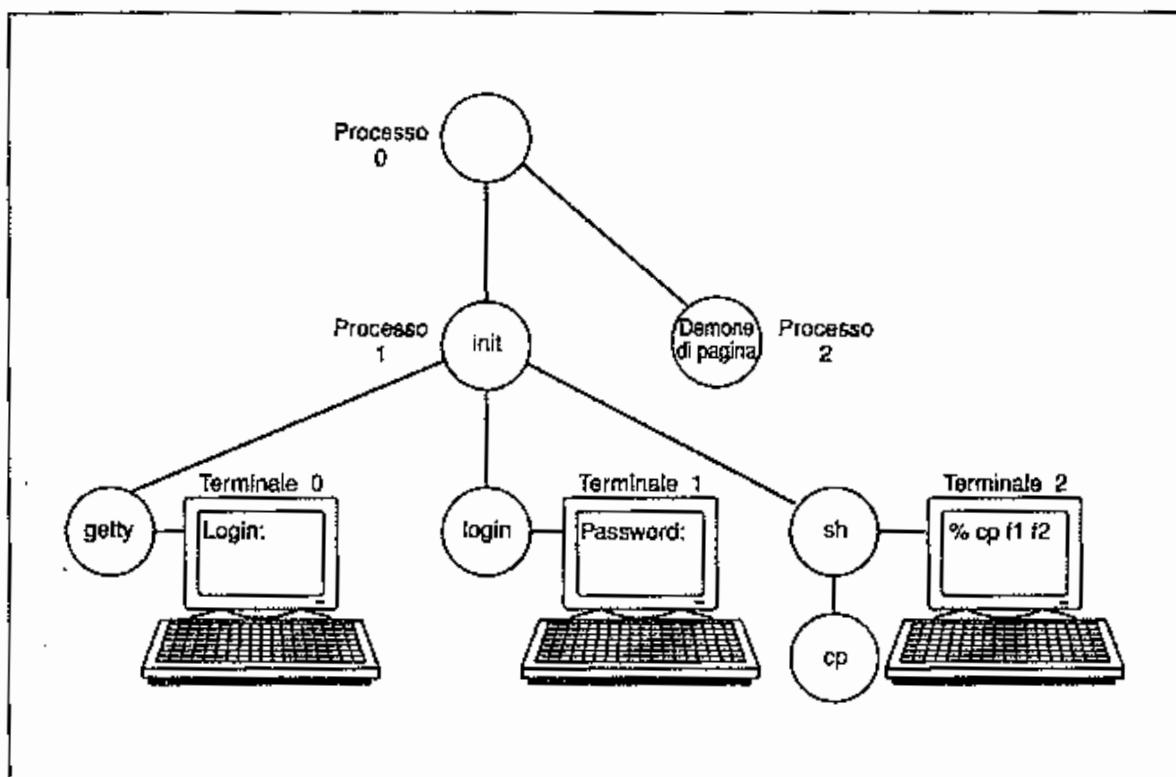


Figura 10.12 La sequenza di processi usata per l'avvio di alcuni sistemi UNIX.

## 10.4 Il modello di memoria di UNIX

Il modello di memoria di UNIX è molto semplice, proprio per rendere i programmi portabili e per rendere possibile l'implementazione di UNIX su macchine con unità di gestione della memoria molto differenti tra loro, passando dalla completa mancanza di gestione (vedi il PC-IBM originale) ad una sofisticata paginazione. In quest'area, il progetto è sostanzialmente lo stesso da decenni, e funziona così bene che non c'è mai stato bisogno di rivederlo. Esamineremo adesso il modello e come viene implementato.

### 10.4.1 Concetti di base

Ogni processo UNIX ha uno spazio degli indirizzi formato da tre segmenti: il testo, i dati e lo stack. La Figura 10.13(a) dà un esempio di spazio degli indirizzi di un processo; il segmento testo (text segment) contiene le istruzioni macchina che formano il codice eseguibile del programma. Esso è prodotto dal compilatore e dall'assemblatore che traducono il C, il C++, o altri linguaggi, in codice macchina; il segmento testo di solito è in modalità di sola lettura. I programmi automodificanti furono messi al bando negli anni '50 perché erano troppo difficili da capire e da correggere, per cui il segmento testo non cresce né si contrae né si modifica in alcun modo.

Il segmento dati (data segment) contiene la memoria per le variabili del programma, le stringhe, gli array o altri dati, ed è formato da due parti: i dati inizializzati e quelli non inizializzati; per ragioni storiche, questi ultimi sono conosciuti come BSS. La parte inizializzata del segmento dati contiene le variabili e le costanti di compilazione che hanno bisogno di un valore iniziale quando il programma parte.

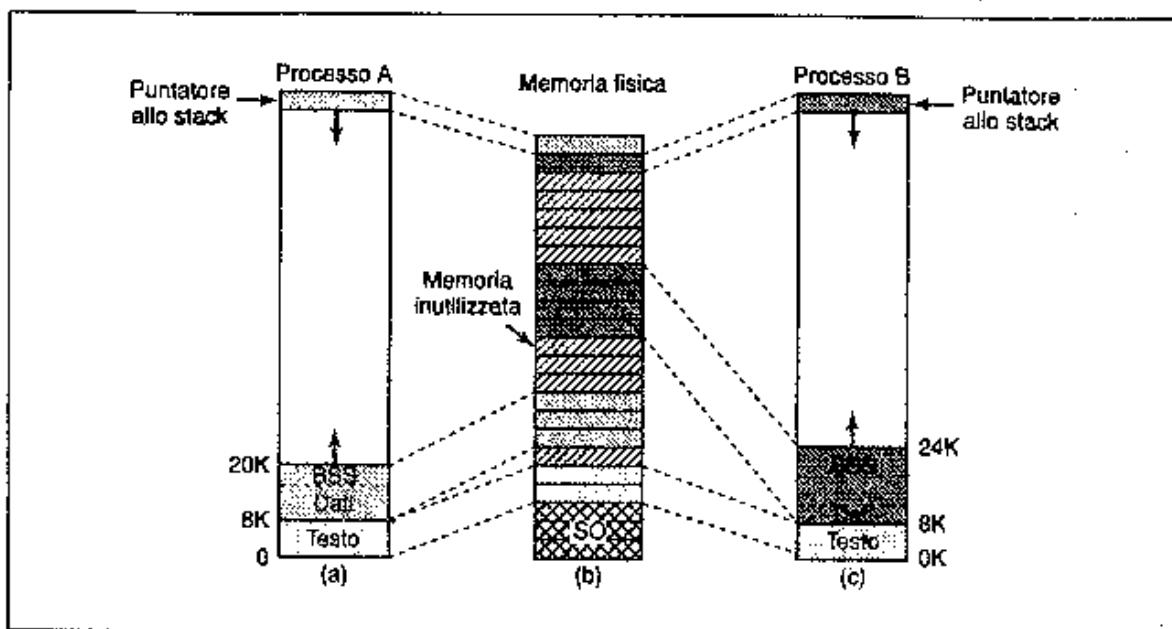


Figura 10.13 (a) Lo spazio degli indirizzi virtuale del processo A. (b) La memoria fisica. (c) Lo spazio degli indirizzi virtuale di B.

Per esempio, in C è possibile dichiarare e contemporaneamente inizializzare un stringa di caratteri: quando il programma parte, si aspetta che la stringa abbia il suo valore iniziale. Per implementare questo costrutto, il compilatore assegna alla stringa un posto nello spazio degli indirizzi ed assicura che, quando il programma è partito, questa localizzazione abbia la stringa appropriata. Dal punto di vista del sistema operativo, i dati inizializzati non sono diversi dal testo del programma, infatti entrambi contengono configurazioni di bit prodotte dal compilatore che devono essere caricate in memoria quando parte il programma.

L'esistenza di dati non inizializzati è in effetti solo un'ottimizzazione: quando una variabile globale non è inizializzata esplicitamente, la semantica del linguaggio C dice che il suo valore iniziale è 0, in pratica la maggior parte delle variabili globali non sono inizializzate ed il valore a loro assegnato è zero. Ciò si potrebbe implementare avendo semplicemente una parte di file binario eseguibile esattamente uguale al numero di byte dei dati, tutti inizializzati, compresi quelli che per difetto hanno il valore zero; ma ciò non si fa, per risparmiare spazio nel file eseguibile.

Il file contiene tutte le variabili inizializzate esplicitamente dopo il testo del programma; e le variabili non inizializzate sono tutte raccolte insieme dopo quelle inizializzate, così tutto quello che il compilatore deve fare è mettere una parola nell'intestazione che dice quanti byte allocare.

Per capire meglio questo punto, si osservi meglio la Figura 10.13(a), in cui il testo del programma occupa 8KB di memoria, ed anche i dati inizializzati occupano 8KB, mentre i dati non inizializzati (BSS) occupano 4KB. Il file eseguibile è di soli 16KB (testo + dati inizializzati), più una piccola intestazione che dice al sistema di allocare altri 4KB dopo i dati inizializzati, ed azzerarli prima di far partire il programma. Questa precauzione evita di memorizzare 4KB di memoria pieni di zero nel file eseguibile.

Diversamente dal segmento di testo, che non può cambiare, il segmento dei dati può cambiare: i programmi cambiano i valori delle variabili in continuazione. Inoltre molti programmi hanno bisogno di allocare lo spazio dinamicamente, durante l'esecuzione. UNIX gestisce ciò permettendo al segmento dei dati di crescere e restringersi a seconda

che la memoria sia allocata o deallocata: è disponibile una chiamata di sistema, brk, che permette ad un programma di definire la dimensione del suo segmento di dati. La procedura di libreria del C, *malloc*, usata di solito per allocare la memoria, usa pesantemente questa chiamata di sistema.

Il terzo segmento è il segmento che contiene lo stack, che, su diverse macchine, si trova all'estremo superiore (o quasi) dello spazio degli indirizzi virtuali, e cresce verso lo zero. Se lo stack raggiunge il fondo dello spazio riservatogli, di solito si ha un errore hardware ed il sistema operativo abbassa il fondo di una pagina. I programmi non gestiscono esplicitamente la dimensione del segmento di stack.

Quando un programma inizia l'esecuzione, il suo stack non è vuoto, ma contiene tutte le variabili di ambiente (shell) insieme con la linea di comando digitata per chiamare il programma. In questo modo un programma può accedere ai suoi argomenti; per esempio, quando si digita il comando

```
cp src dest
```

il programma *cp* inizia il suo lavoro avendo nello stack la stringa "cp src dest", e in questo modo può trovare i nomi del file sorgente e del file di destinazione. La stringa viene rappresentata come un vettore di puntatori ai simboli nella stringa in modo da facilitarne l'analisi.

Quando due utenti stanno utilizzando lo stesso programma, per esempio l'editor, potrebbe essere possibile, ma inefficiente, tenere due copie in memoria del suo testo contemporaneamente. Diversi sistemi UNIX utilizzano quindi la tecnica dei segmenti di testo condivisi (shared text segment): nella Figura 10.13(a) e nella Figura 10.13(c) si vedono due processi che hanno lo stesso segmento di testo. Nella Figura 10.13(b) si vede una possibile organizzazione della memoria fisica, dove entrambi i processi condividono lo stesso pezzo di testo. La corrispondenza viene fatta mediante l'hardware di gestione della memoria virtuale.

I segmenti dei dati e dello stack non sono mai condivisi, eccetto che dopo una *fork*, ed anche allora solo per le pagine non modificate; se uno di essi avesse bisogno di ulteriore

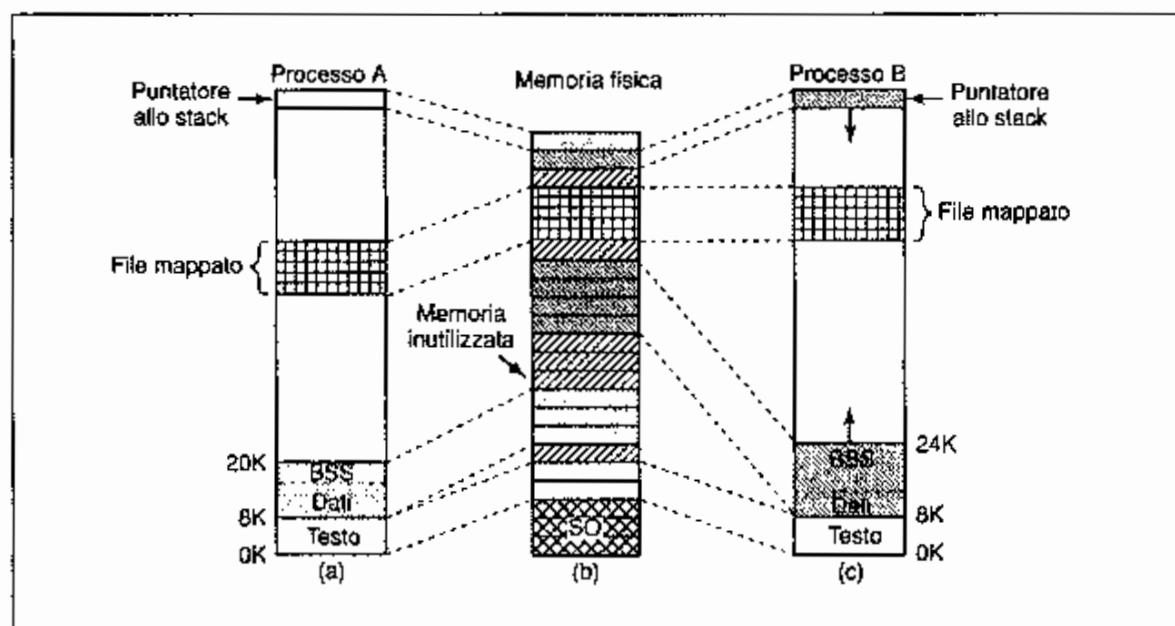


Figura 10.14 Due processi possono condividere uno stesso file mappato.

spazio e nella memoria adiacente non vi fosse spazio sufficiente, non c'è problema, perché le pagine virtuali adiacenti possono anche corrispondere a pagine fisiche non adiacenti.

Su alcuni computer, l'hardware fornisce spazi degli indirizzi separati per le istruzioni e i dati, e in questo caso UNIX può gestirli. Per esempio, se tale caratteristica è disponibile su un computer con indirizzi a 32 bit, vi saranno 2<sup>32</sup> bit di spazio degli indirizzi per istruzione e 2<sup>32</sup> bit addizionali nello spazio degli indirizzi condivisi fra i segmenti dati ed i segmenti di stack. Un salto alla posizione 0 porta all'indirizzo 0 dello spazio testo, e una MOVE da 0 usa l'indirizzo 0 nello spazio dati. In questo modo si duplica lo spazio degli indirizzi disponibile.

Molte versioni di UNIX supportano i file mappati in memoria che vengono mappati in una porzione dello spazio degli indirizzi del processo, in modo da poter essere letti e scritti come se fossero un array di byte in memoria. La mappatura di file rende gli accessi casuali più facili delle chiamate I/O come `read` e `write`: così vengono organizzate le librerie condivise in UNIX. In Figura 10.14 si può osservare un file mappato da due processi nello stesso momento, ed a diversi valori dello spazio degli indirizzi.

Un vantaggio ulteriore della mappatura dei file è che due o più processi possono mappare lo stesso file simultaneamente, e le scritture sul file da parte di un processo sono immediatamente visibili a tutti gli altri. In effetti, mappare un file temporaneo (che verrà cancellato al termine dei processi) permette a più processi di condividere memoria a banda larghissima. Come caso più estremo si consideri il caso in cui due o più processi mappino un file che copre l'intero spazio di indirizzi, fornendo una forma di condivisione che è a metà strada tra i processi separati ed i thread. Qui, lo spazio degli indirizzi è condiviso (come nei thread), ma ogni processo mantiene il proprio file aperto, ed i propri segnali, cosa che non avviene per i thread. In pratica, comunque non avviene mai che due spazi degli indirizzi siano esattamente gli stessi.

## 10.4.2 Chiamate di sistema per la gestione della memoria in UNIX

POSIX non specifica alcuna chiamata di sistema per la gestione della memoria perché questa parte era considerata troppo dipendente dalla macchina per la standardizzazione. Il problema fu così nascosto sotto il tappeto, dicendo che i programmi che avessero avuto bisogno della gestione dinamica della memoria avrebbero potuto usare la procedura di libreria `malloc` (definita dallo standard ANSI C). L'implementazione della `malloc` è perciò rimasta al di fuori della portata dello standard POSIX. In altri ambienti, questo approccio è noto come il gioco dello scarica barile.

In pratica, molti sistemi UNIX hanno una chiamata di sistema, spesso chiamata `brk`, che specifica la dimensione di cui necessita il segmento dati (vedere Figura 10.15). La chiamata `brk` specifica la dimensione del segmento dati dando l'indirizzo del primo byte

| Chiamata di sistema                                     | Descrizione                               |
|---------------------------------------------------------|-------------------------------------------|
| <code>s = brk(ind)</code>                               | Cambia la dimensione del segmento di dati |
| <code>a = mmap(ind, len, prot, flag, fd, offset)</code> | Mappa un file                             |
| <code>s = munmap(ind, len)</code>                       | Toglie la mappatura da un file            |

Figura 10.15 Alcune chiamate di sistema relative alla gestione della memoria. Il codice `s` restituito è -1 in caso di errore; `a` e `ind` sono indirizzi di memoria, `len` è la lunghezza, `prot` controlla la protezione, `flag` sono bit generici, `fd` è il descrittore di file ed, infine, `offset` è l'offset del file.

oltre esso; se il nuovo valore è più grande del vecchio, il segmento dei dati diventa più grande, se è più piccolo il segmento si restringe.

Le chiamate di sistema `mmap` e `unmap` controllano i file mappati in memoria. `ind`, il primo parametro di `mmap`, determina l'indirizzo di mappatura del file (o di una sua porzione) e deve essere sempre un multiplo della dimensione di una pagina; se questo parametro è 0, il sistema determina da solo l'indirizzo e lo restituisce in `a`. Il secondo parametro, `lun`, descrive quanti byte devono essere mappati, e deve essere anch'esso un multiplo della dimensione di una pagina. Il terzo parametro, `prot`, determina la protezione del file mappato e può assumere il valore di leggibile, scrivibile, eseguibile, e loro combinazioni. Il quarto parametro, `flag`, determina se è un file privato o condivisibile e se `ind` è utile o no. Il quinto parametro, `fd`, è il descrittore del file: solo i file aperti possono essere mappati e, quindi, per mappare un file occorre prima aprirlo. Infine, `offset` descrive dove deve iniziare la mappatura del file, e non è necessario se si desidera iniziare a mappare dal byte 0, altrimenti si deve specificare un limite di pagina.

L'altra chiamata di sistema `unmap` non fa altro che rimuovere la mappatura di un file e, se viene chiamata solo su una porzione di file, il resto dello stesso rimane mappato.

### 10.4.3 Implementazione della gestione della memoria in UNIX

Precedentemente al 3BSD, molti sistemi UNIX erano basati sullo swapping, che funziona come descritto di seguito: quando i processi esistenti erano più di quelli che possono risiedere in memoria centrale, alcuni di essi venivano trasferiti su disco. Poiché un processo era sempre trasferito interamente (tranne che per le porzioni condivise di testo), esso era o in memoria o su disco.

#### Swapping

Il movimento tra la memoria e il disco era gestito dal livello superiore dello scheduler, conosciuto come `swapper`. Lo swapping tra la memoria ed il disco veniva iniziato quando il kernel non aveva memoria sufficiente, per uno dei seguenti eventi:

1. Una chiamata di sistema `fork` ha bisogno di memoria per un processo figlio.
2. Una chiamata di sistema `brk` ha bisogno di espandere un segmento di dati.
3. Uno stack diventa troppo grande e supera lo spazio ad esso riservato.

Inoltre, al momento in cui bisognava portare in memoria un processo che era stato troppo a lungo sul disco, spesso era necessario rimuovere un altro processo per predisporre lo spazio ad esso necessario.

Per scegliere la vittima, lo swapper prima esaminava i processi che erano bloccati in attesa di qualcosa (per esempio input da terminale); infatti è meglio rimuovere un processo impossibilitato a girare che non uno con questa possibilità. Se ne trovava uno, o più d'uno, veniva scelto quello che aveva il valore più alto, ottenuto sommando la priorità al tempo di residenza. Un buon candidato era un processo che aveva consumato molto tempo di CPU, o uno che era stato in memoria troppo a lungo, anche se faceva soprattutto operazioni di I/O. Se non era disponibile nessun processo bloccato, allora si sceglieva un processo pronto basandosi sugli stessi criteri.

Ogni pochi secondi, lo swapper esaminava l'elenco dei processi sul disco per vedere se uno di essi fosse disponibile per essere eseguito; in caso affermativo selezionava quel-

lo che era stato sul disco più a lungo e successivamente controllava se questo modo di procedere dava luogo a uno swap facile o difficile. Uno swap era facile quando si aveva a disposizione sufficiente memoria, per cui non c'era bisogno di rimuovere dalla memoria alcun processo per fare posto a quello nuovo; uno swap difficile richiedeva la rimozione della memoria di uno o più processi. Uno swap facile era implementato portando in memoria il processo, mentre uno swap difficile richiedeva che prima fosse liberata abbastanza memoria da uno swap out di uno o più processi; dopo di ciò veniva portato in memoria il processo prescelto.

Questo algoritmo veniva ripetuto finché non si verificava una delle seguenti condizioni: (1) nessun processo su disco era pronto a girare; (2) la memoria era così piena di processi appena caricati che non c'era alcuno spazio libero. Per impedire il thrashing, nessun processo veniva riportato su disco prima che non avesse trascorso almeno 2 secondi in memoria.

Due liste tenevano traccia degli spazi liberi in memoria e sul disco; così, quando c'era bisogno di spazio, l'algoritmo visitava la lista appropriata e restituiva il primo spazio sufficientemente grande per poter inserire il processo in esame, rimuovendolo dalla lista o riducendone la dimensione.

## Paginazione in UNIX

Tutte le versioni di UNIX per il PDP-11, per le macchine Interdata ed anche le prime implementazioni per il VAX, si basavano sullo swapping appena descritto. A partire da 3BSD, a Berkeley aggiunsero la paginazione per gestire i grossi programmi che stavano scrivendo. Attualmente tutti i sistemi UNIX hanno la paginazione a richiesta, le cui origini affondano nella 3BSD. Di seguito si parlerà del progetto del 4BSD pur tenendo presente che il System V non è molto diverso da esso.

L'idea alla base della paginazione del 4BSD è semplice: un processo non ha bisogno di stare tutto in memoria per girare, infatti tutto ciò che è richiesto per l'esecuzione è la struttura dell'utente e la tabella delle pagine. Se questi due oggetti sono in memoria, il processo può essere considerato "in memoria" e schedulato per girare: le pagine del testo, i dati e i segmenti di stack sono portati dinamicamente in memoria, uno alla volta, quando vengono richiesti. Se la struttura utente e la tabella delle pagine non sono in memoria, il processo non può essere eseguito finché lo swapper non ve le porta.

Lo UNIX di Berkeley non usa il modello degli insiemi di lavoro, né altre forme di pre-paginazione perché, per fare ciò, è necessaria la conoscenza di quali sono le pagine che si stanno o meno usando. Questa informazione non è facilmente acquisibile nel VAX poiché non è fornito di bit di riferimento alle pagine (sebbene essa si possa ottenere a livello software con un overhead non indifferente).

La paginazione è implementata parzialmente dal kernel e parzialmente da un nuovo processo chiamato demone di pagina (page daemon), il quale è il processo 2 mentre il processo 0 è lo swapper ed il processo 1 è init (vedere la Figura 10.12). Come tutti i demoni, il demone di pagina è fatto partire periodicamente, così può guardare in giro se c'è qualcosa da fare; se scopre che il numero di pagine libere in memoria è troppo basso, inizia a prendere qualche provvedimento per liberarne qualcuna.

In 4BSD la memoria principale è organizzata come mostrato in Figura 10.16. È formata da tre parti: le prime due contengono il kernel e la mappa della memoria fisica (core map) e sono permanentemente in memoria (non sono mai paginate su memoria di massa); la parte rimanente viene divisa in pagine fisiche (page frame), ognuna delle quali può contenere il testo, i dati, una pagina di stack, una pagina della tabella delle pagine o essere in lista libera.

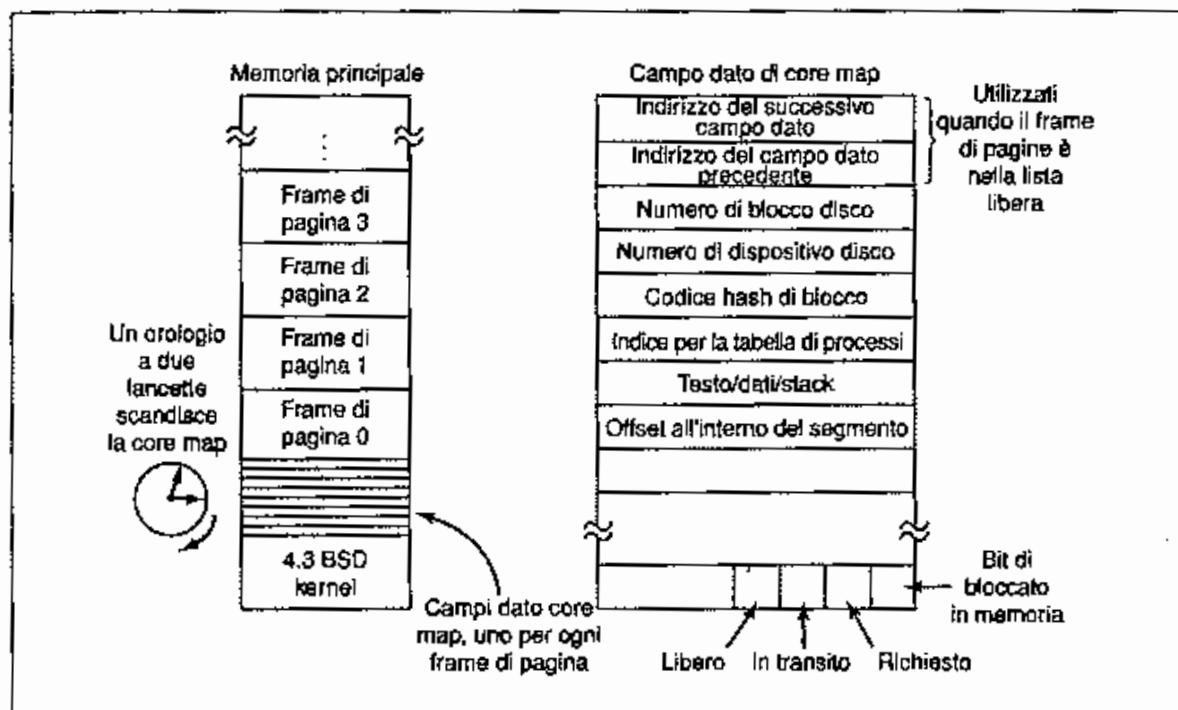


Figura 10.16 La core map in 4BSD.

La **core map** contiene l'informazione sul contenuto delle pagine fisiche: il suo elemento 0 descrive la pagina fisica 0, l'elemento 1 descrive la pagina fisica 1, e così via. Con pagine da 1KB e campi dato da 16 byte, la core map occupa meno del 2% della memoria. Le prime due locazioni di un elemento della core map (Figura 10.16) vengono usati solo quando la pagina fisica corrispondente è nella lista libera, per realizzare una lista bidirezionale che tiene insieme tutte le pagine libere; le tre locazioni successive vengono usate quando la pagina fisica contiene informazioni. Ad ogni pagina in memoria è riservato uno spazio su disco dove essa viene salvata quando deve fare spazio ad un'altra pagina, per cui queste informazioni si usano per trovare la locazione della pagina memorizzata su disco. Le successive tre locazioni danno gli elementi della tabella dei processi relative al processo che si trova in quella pagina, il segmento e la sua posizione; l'ultimo elemento contiene alcuni flag necessari all'algoritmo di paginazione.

Quando un processo parte, si può avere un fault di pagina, perché le sue pagine non sono in memoria; il sistema prende la prima pagina fisica dalla lista libera, la rimuove e carica al suo interno la pagina richiesta; nel caso la lista libera sia vuota, il processo è sospeso finché il demone di pagina non ha liberato una pagina fisica.

## L'algoritmo di rimpiazzamento di pagina

L'algoritmo di rimpiazzamento di pagina viene eseguito dal demone di pagina; questi viene attivato ogni 250 ms per controllare se il numero di pagine fisiche libere corrisponde al parametro di sistema chiamato *latsfree* (di solito pari a 1/4 della memoria). Se non ci sono abbastanza pagine libere, il demone inizia il trasferimento dalla memoria sul disco finché non si hanno *latsfree* pagine fisiche libere; al contrario, se il demone scopre che ci sono più pagine libere di quelle indicate dal parametro suddetto, non avendo nulla da fare ritorna inattivo. Se la macchina ha sufficiente memoria e pochi processi attivi il demone di pagina resta quasi sempre inattivo.

Il demone di pagina usa una versione modificata dell'algoritmo dell'orologio; esso è un algoritmo globale che non tiene conto del processo a cui appartiene la pagina che è stata rimossa, per cui il numero delle pagine assegnate ad un certo processo varia di volta in volta.

Questo algoritmo opera esaminando le pagine in maniera circolare (come se fossero sul quadrante dell'orologio): nel primo passo, quando la lancetta punta ad una pagina, viene azzerato il bit di uso; nel secondo passo, le pagine che non sono state utilizzate dopo il primo passo avranno il bit di uso azzerato, e saranno messe nella lista libera (dopo averne eventualmente trasferito su disco il contenuto, se erano sporche). Ogni pagina nella lista libera mantiene il suo contenuto, che si può recuperare nel caso sia riutilizzata.

In macchine come il VAX che non hanno il bit di uso, quando la lancetta dell'orologio punta ad una pagina nel primo passo, viene azzerato il bit di uso software e la pagina viene segnata come non valida nella tabella delle pagine. Quando si accede alla pagina nuovamente, si ha un fault di pagina, il che permette al sistema operativo di porre a 1 il bit di uso software. Si ha lo stesso effetto del bit di uso hardware, ma l'implementazione è molto più costosa: il costo di una progettazione hardware scadente ricade sul software.

All'inizio, lo UNIX di Berkeley usava l'algoritmo dell'orologio, ma in seguito si scoprì che con memorie di grandi dimensioni esso prendeva troppo tempo, per cui fu modificato nell'algoritmo dell'orologio a due lancette, come mostrato a sinistra in Figura 10.16. In questo algoritmo, il demone di pagina gestisce due puntatori alla core map; quando esso è in funzione, prima azzera il bit di uso della pagina puntata dalla prima lancetta, dopo controlla il bit di uso della seconda lancetta, solo dopo aver fatto ciò, fa avanzare le lancette. Quando le due lancette sono vicine, solo le pagine realmente usate in modo pesante rischiano di essere utilizzate tra i due passaggi, se le due lancette sono a 359 gradi (cioè la seconda lancetta è appena avanti all'altra) si ritorna all'algoritmo dell'orologio originale. Ogni volta che gira il demone di pagina, le lancette ruotano per meno di un intero giro; la loro rotazione dipende da quanto ci mettono a portare il numero delle pagine nella lista libera al valore *lotsfree*.

Se il sistema osserva che la frequenza di paginazione è troppo alta ed il numero delle pagine libere è sempre molto al di sotto di *lotsfree*, lo swapper inizia a rimuovere uno o più processi dalla memoria affinché essi non debbano competere più per le pagine fisiche. L'algoritmo di swap out del 4BSD funziona come segue: controlla se qualcuno dei processi è stato inattivo per 20 secondi o più, nel qual caso quello che è stato inattivo più a lungo viene scaricato; se non ce n'è nessuno, considera i quattro processi più grandi in memoria, e rimuove quello che è in memoria da più tempo; se necessario, questo algoritmo viene ripetuto finché non è stata recuperata sufficiente memoria.

Ogni pochi secondi, lo swapper controlla l'esistenza su disco di qualche processo in stato di pronto da portare in memoria. Ad ogni processo su disco è assegnato un valore che è una funzione del tempo passato dal momento in cui esso è stato scaricato, della sua dimensione, del valore assegnato con una eventuale chiamata nice e del tempo in cui è rimasto inattivo prima di essere scaricato. La funzione è poi pesata per portare in memoria di solito il processo che è stato fuori più a lungo, a meno che esso non sia troppo grande. La teoria dice che portare in memoria un grosso processo è costoso, per cui non dovrebbe essere mosso troppo spesso; lo si porterà in memoria solo se ci sono abbastanza pagine libere, cosicché al momento in cui si avrà l'inevitabile fault di pagina, ci saranno pagine fisiche disponibili per esso. Lo swapper porta effettivamente in memoria solo la struttura utente e la tabella delle pagine mentre il testo, i dati e le pagine dello stack vengono caricate solo al momento in cui sono usate.

Ogni segmento di un processo attivo ha un posto sul disco dove risiede quando è scaricato sia con la paginazione che con lo swapping. I segmenti dei dati e dello stack van-

no su un dispositivo di lavoro, ma il testo del programma viene paginato dal file binario eseguibile; non si usano copie su file temporanei per il testo.

La paginazione nel System V è fondamentalmente simile a quella del 4BSD, il che non sorprende poiché la versione di Berkeley ha funzionato bene per anni prima che la paginazione fosse adottata anche nel System V, ciò non di meno ci sono due differenze interessanti.

La prima consiste nel fatto che System V usa l'algoritmo dell'orologio originale ad una lancetta, invece che quella a due lancette; inoltre, invece di mettere nella lista libera una pagina non usata, essa vi viene inserita solo se non è stata usata per un numero  $n$  di passi consecutivi. Benché questa scelta non renda le pagine libere velocemente come nella versione di Berkely, esso aumenta in modo sostanziale la possibilità che una pagina, una volta liberata, non sarà necessaria di nuovo in tempi brevi.

La seconda differenza consiste nel fatto che System V utilizza due variabili *min* e *max* al posto della sola variabile *lotsfree*: quando il numero delle pagine libere scende al di sotto del valore di *min*, il demone delle pagine inizia a liberare pagine, e continua a liberarne finché non ha raggiunto il valore *max*. Questo approccio elimina la potenziale instabilità del 4BSD. Si consideri una situazione in cui il numero di pagine libere è una meno di *lotsfree*, per cui entra in funzione il demone di pagina portandole al valore di *lotsfree*; se dopo si ha un altro fault di pagina, dovuto all'uso di una pagina che riduce di nuovo *lotsfree*, viene provocata la riattivazione del demone. L'introduzione di *min* e *max* abbastanza distanti fra loro assicura che il demone, al momento della sua attivazione, predisporrà abbastanza pagine libere da non dover essere riattivato per una quantità di tempo significativa.

## La gestione della memoria in Linux

Ogni processo in Linux su una macchina a 32-bit ottiene 3GB di spazio degli indirizzi virtuale, con 1GB riservato per le tabelle delle pagine ed altri dati del kernel. Il GB del kernel non è visibile quando il processo gira in modo utente, ma diventa accessibile quando il processo passa in modo kernel. Lo spazio degli indirizzi viene creato al momento in cui il processo viene creato ed è sovrascritto dalle chiamate della funzione di sistema exec.

Lo spazio degli indirizzi virtuale si divide in aree o regioni omogenee, contigue, ed allineate alle pagine. Quindi, ogni area è fatta da una sequenza di pagine consecutive con la stessa protezione e le stesse proprietà di paginazione. Il segmento testo e i file mappati possono considerarsi esempio di aree (si veda la Figura 10.14), e fra le aree, nello spazio degli indirizzi virtuale, vi possono essere buchi. Il risultato di un riferimento ad un buco è un fault di pagina fatale; inoltre, la dimensione della pagina è fissa, ad esempio 4KB per il Pentium e 8KB per Alpha.

Nel kernel, ogni area viene descritta da un campo dato chiamato *vm\_area\_struct*: tutti questi campi dato di un processo sono collegati tra loro in una lista ordinata per indirizzi virtuali in modo da trovare tutte le pagine, e quando la lista cresce troppo (più di 32 campi dato), viene creato un albero per accelerare le operazioni di ricerca. Il campo dato *vm\_area\_struct* elenca le proprietà dell'area associata, e cioè: il modo di protezione (ad esempio se è solo leggibile o se è possibile leggere e scrivere sull'area), se l'area è paginabile o meno, e in quale direzione cresce (su per segmenti di dati, o giù per lo stack).

Il campo dato inoltre memorizza se l'area è privata per un processo o può essere condivisa tra due o più processi. Dopo una operazione di fork, Linux fa una copia di una lista di aree per il processo figlio, ma fa in modo che le liste del padre e del figlio puntino alle stesse tabelle di pagine. Le aree sono marcate come leggibili e scrivibili ma le pagine sono marcate come solo leggibili. Se un processo (padre o figlio) cerca di scrivere su

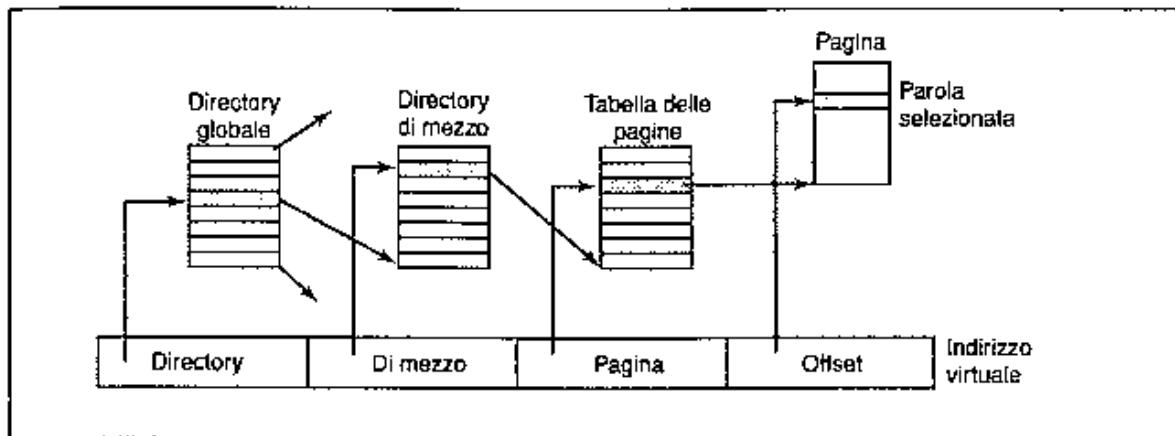


Figura 10.17 Linux usa tabelle di pagine a tre livelli.

una pagina, avviene un fault di protezione, ed il kernel capisce che essendo l'area scrivibile è la pagina no, occorre fare una copia della pagina per il processo che ha richiesto la scrittura, marcata `read/write`. Questo meccanismo realizza la `copy-on-write`.

Inoltre, il campo dato `vm_area_struct` registra se l'area ha uno spazio su disco, e dove. I segmenti testo usano l'eseguibile binario come copia su disco, ed i file mappati in memoria il corrispondente file disco; le altre aree, come gli stack, non hanno backing storage assegnato fino a quando non vengono paginati fuori.

Linux utilizza uno schema di paginazione a tre livelli. Sebbene questo schema sia stato messo nel sistema per Alpha, può essere utilizzato (in forma degenere) in tutte le altre architetture. Ogni indirizzo virtuale viene diviso in quattro campi, come mostrato in Figura 10.17; il campo directory viene utilizzato come indice nella directory globale, che è presente in ogni processo. Il valore trovato rappresenta un puntatore ad una delle tabelle di pagine di mezzo, che viene ulteriormente indicizzata con un campo dall'indirizzo virtuale. Il campo dato selezionato punta alla tabella di pagine finale, che è indicizzata dal campo pagina dell'indirizzo virtuale. Il campo dato trovato punta alla pagina necessaria. Sul Pentium, che utilizza la paginazione a due livelli, la tabella di pagina di mezzo possiede un unico campo dato, cosicché il campo dato della directory globale sceglie praticamente la tabella di pagina da usare.

La memoria fisica, invece, viene utilizzata per vari scopi: il kernel ne fa largo uso; non esistono parti del kernel che vengono paginate fuori. Il resto della memoria viene utilizzata per le pagine utenti, la cache di buffer utilizzata dal file system, la cache di paginazione ed altro. La cache di buffer mantiene blocchi di file che sono stati recentemente letti, o che sono stati letti in anticipo, in attesa di essere utilizzati nell'immediato futuro; essa è dinamica in dimensioni e compete per la stessa pool di pagine delle pagine utente. La cache di paginazione non è una vera e propria cache separata, ma semplicemente un insieme di pagine utente che non sono più necessarie e che stanno attendendo di essere paginate fuori. Se una pagina nella cache di paginazione viene riusata prima di essere eliminata dalla memoria, può essere riassegnata velocemente.

Inoltre, Linux supporta moduli caricati dinamicamente, solitamente driver di dispositivi, che possono essere di dimensione arbitraria, ed ognuno deve essere allocato in un pezzo contiguo di memoria kernel. Come conseguenza, Linux gestisce la memoria fisica in modo tale da poterne acquisire un pezzo di dimensione arbitraria in ogni momento. L'algoritmo è detto **algoritmo buddy** (**algoritmo dei compagni**), e viene di seguito descritto.

L'idea di base per la gestione di un pezzo di memoria è la seguente: inizialmente la memoria si compone di un solo pezzo contiguo, 64 pagine nell'esempio in Figura 10.18 (a).

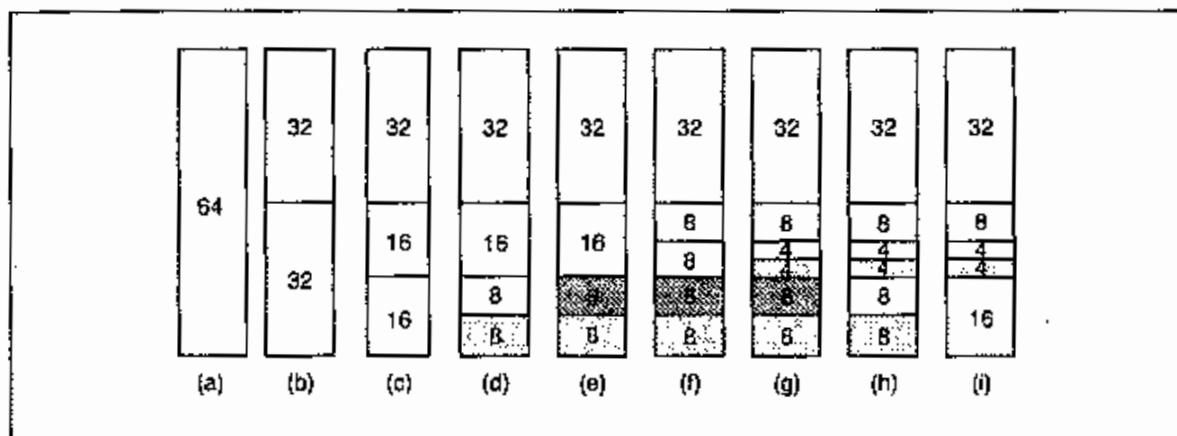


Figura 10.18 Le fasi dell'algoritmo dei compagni.

Al momento di una richiesta di memoria, questa viene arrotondata ad una potenza di due, ad esempio 8 pagine. L'intero pezzo di memoria viene diviso in due come mostrato in (b); se ogni pezzo risulta ancora troppo grande, il pezzo più basso di memoria viene diviso nuovamente in due (c) ed ancora (d). Nell'esempio si è ottenuto un pezzo di dimensione corretta, che viene allocato per il chiamate, come mostrato dal rettangolo ombreggiato in (d).

Si supponga adesso che una nuova richiesta di 8 pagine arrivi al sistema: questa può essere soddisfatta immediatamente (e). A questo punto arriva una terza richiesta di 4 pagine: il pezzo più piccolo disponibile viene diviso (f) e la metà più bassa viene concessa (g). Al passo successivo il secondo pezzo di 8 pagine viene rilasciato (h), così come al passo successivo avviene per un altro pezzo di 8 pagine. Poiché due pezzi da 8 pagine, contigui, risultano liberi, essi possono essere uniti nuovamente per diventare un unico blocco da 16 pagine (i).

Linux gestisce la memoria utilizzando l'algoritmo buddy con in aggiunta un array che contiene come primo elemento la testa di una lista di blocchi di dimensione 1 unità, come secondo elemento la testa di una lista di blocchi di 2 unità, come successivo elemento la testa di una lista di blocchi di dimensione 4, eccetera. In questo modo, ogni blocco di dimensione una potenza di due può essere trovato in modo veloce.

Questo algoritmo costringe ad una eccessiva frammentazione della memoria, dato che, ad esempio, per allocare memoria per un pezzo di 65 pagine occorre un pezzo di 128 pagine.

Per cercare di alleviare il problema, Linux possiede una seconda allocazione di memoria che prende i pezzi di memoria ottenuti dall'algoritmo buddy e li divide in pezzi più piccoli. Un terzo allocatore di memoria è presente solo quando la memoria richiesta deve essere contigua nello spazio virtuale, ma non nella memoria fisica. Si noti che tutti questi allocator sono in un certo senso ereditati da System V.

Linux è un sistema a paginazione su richiesta (demand-page) e non possiede alcuna operazione di prepaginazione né insiemi di lavoro (sebbene vi sia una chiamata di sistema nella quale un utente può suggerire che una pagina potrà servire da lì a breve, nella speranza che arrivi immediatamente quando realmente necessaria). I segmenti testo ed i file mappati sono paginati sui rispettivi file su disco; tutto il resto è paginato o sulla partizione di paginazione, se presente, o su un file di paginazione di lunghezza fissa (ce n'è al massimo 8). I file di paginazione hanno una certa priorità e possono essere dinamicamente aggiunti e rimossi. La paginazione in una partizione separata, alla quale si può accedere come un semplice dispositivo, è più efficiente: in primo luogo, la mappatura tra

blocchi di file e blocchi disco non è necessaria (si risparmiano le letture dei blocchi indiretti dal disco); in secondo luogo, le scritture fisiche possono essere di qualsiasi dimensione e non solo della dimensione dei blocchi di file. Infine, una pagina viene sempre scritta in modo contiguo su disco, mentre con un file di paginazione questo potrebbe non accadere.

Linux non alloca le pagine sul dispositivo di paginazione o sulla partizione se non è necessario. Ogni dispositivo o file incomincia con una mappa di bit che mostra quali pagine sono libere: quando una pagina deve essere buttata fuori dalla memoria, la partizione di paginazione o il file con priorità più alta, che ha ancora spazio viene scelto per l'allocazione della pagina in questione. Normalmente, la partizione di paginazione, se presente, possiede sempre priorità maggiore di ogni altro file di paginazione. Infine, la locazione disco viene scritta nella tabella delle pagine.

La sostituzione delle pagine avviene come segue: Linux cerca di mantenere sempre qualche pagina libera in modo da poterla concedere su richiesta, e naturalmente questa pool deve essere riempita in continuazione, ed è proprio questo che l'algoritmo di paginazione fa. Al momento dell'avvio del sistema, il processo *init* fa partire un demone di pagina, *kswapd* che si mette in esecuzione ogni secondo: in pratica controlla se vi sono abbastanza pagine libere in quel momento e, se sì, si riaddormenta per un altro secondo, a meno che una richiesta di pagine non lo svegli. Il codice di questo demone è formato da un loop eseguito 6 volte con urgenza crescente. Perché 6 volte? Probabilmente perché all'autore in quel momento 4 sembravano troppo poche ed 8 troppe: Linux spesso è fatto così.

Il corpo del loop della procedura non fa altro che chiamare tre diverse procedure, ognuna delle quali è impegnata a ricercare pagine che possono essere liberate in vari settori. Il valore di urgenza della richiesta viene passato come parametro alle procedure ed in base a questo, le pagine da liberare vengono richieste con maggiore insistenza: ciò di solito si traduce in quante pagine varrà ispezionate prima di terminare. Questa politica permette di recuperare le pagine che possono essere liberate facilmente in ogni categoria, prima di dover attuare una ricerca tra le pagine che sono un po' più difficili da liberare. Se le pagine rilasciate secondo questo procedimento bastano, il demone ritorna a dormire.

Vediamo adesso in dettaglio come operano le tre procedure chiamate nel loop del demone: la prima interroga la cache di paginazione e la cache di buffer del file system, in cerca di pagine non usate di recente, utilizzando un algoritmo tipo quello del clock. La seconda procedura dà un occhio se vi sono pagine condivise non più molto utilizzate da nessuno; la terza, infine, cerca di recuperare le pagine ordinarie degli utenti ed è la più interessante e per questo degna di nota.

In primo luogo, in questa procedura viene fatto un loop su tutti i processi per vedere quale possiede più pagine in memoria. Una volta localizzato il processo, vengono esaminate tutte le sue *vm\_area\_struct* e vengono analizzate tutte le pagine in ordine di indirizzo virtuale, incominciando da dove ci si era fermati la volta precedente. Se si trova una pagina non valida, assente dalla memoria, condivisa, bloccata in memoria oppure utilizzata per DMA, viene saltata. Se la pagina ha il reference bit attivo, viene disattivato e la pagina viene resa disponibile; se il bit di riferimento è, invece, inattivo, la pagina va sul disco: questa parte dell'algoritmo è simile a quella del clock (eccetto per il fatto che la pagina non vengono scandite secondo un ordine FIFO).

Se una pagina viene scelta per eliminarla ed è pulita, viene abbandonata immediatamente. Se è sporca e possiede una pagina per salvarla su disco, viene schedulata per una scrittura a disco. Infine, se è sporca e non ha pagina di *backing store*, viene spostata nella cache di paginazione, dalla quale potrebbe anche venire tolta, se fosse riutilizzata prima di metterla effettivamente su disco. L'idea di cercare tra le pagine seguendo l'ordine degli indirizzi virtuali è che si spera che pagine vicine rispetto agli indirizzi virtuali deb-

bano essere utilizzate in gruppo, e quindi scritte a disco come gruppo e portate in memoria sempre come gruppo.

Un altro aspetto del sistema di gestione della memoria è un secondo demone che ancora non abbiamo analizzato: `hdflush`. Tale demone si sveglia periodicamente (ed in alcuni casi viene esplicitamente svegliato) per controllare se esistono troppe pagine sporche. Se questo accade, incomincia a scriverle a disco.

## 10.5 Input/Output in UNIX

Il sistema di I/O in UNIX è abbastanza banale. In pratica i dispositivi di I/O sono trattati come file ai quali si accede mediante chiamate `read` e `write`, le stesse che sono utilizzate per accedere ai file tradizionali. Solo in alcuni casi devono essere definiti alcuni parametri delle chiamate di sistema, e per questo occorre una speciale chiamata di sistema UNIX. Studieremo questi argomenti nelle prossime sezioni.

### 10.5.1 Concetti di base

Come tutti i computer, quelli su cui gira UNIX hanno connessi dei dispositivi di I/O come dischi, stampanti e reti, ed è necessario un modo per permettere ai programmi di accedere a questi dispositivi. Sebbene siano possibili varie soluzioni, quella di UNIX è di integrarli nel file system sotto il nome di file speciali. Ad ogni dispositivo di I/O viene assegnato un pathname, di solito `/dev`. Per esempio, la stampante può essere `/dev/lp`, il disco può essere `/dev/hd1` e la rete `/dev/net`.

A questi file speciali si può accedere nello stesso modo di qualsiasi altro file, senza comandi speciali o chiamate di sistema; le chiamate di sistema `read` e `write` andranno benissimo. Per esempio, il comando

```
cp file /dev/lp
```

copia `file` sulla stampante, producendone la stampa (supponendo che in quel momento essa sia permessa). I programmi possono aprire, leggere e scrivere i file speciali nello stesso modo dei file normali; infatti, il programma `cp` dell'esempio precedente non è a conoscenza che sta stampando. In questo modo non è necessario alcun meccanismo speciale per l'I/O.

I file speciali vengono divisi in due categorie, a blocco e a carattere. Un file speciale a blocchi è formato da una sequenza di blocchi numerati; la proprietà importante di questo tipo di file è che si può accedere e si può indirizzare in modo individuale ciascun blocco. In altre parole un programma può aprire un file speciale a blocchi e leggere, per esempio, il blocco 124 senza aver letto in precedenza i blocchi da 0 a 123. Questi file sono tipicamente usati per i dischi.

I file speciali a caratteri sono usati di solito per i dispositivi i cui input o output sono flussi di caratteri. Le tastiere, le stampanti, le reti, i mouse, i plotter e molti altri dispositivi di I/O che accettano o producono dati per gli utenti usano i file speciali a caratteri. Non è possibile (e neanche sensato) accedere al blocco 124 su un mouse.

Associato ad ognuno di questi file speciali vi è un driver responsabile della gestione del dispositivo associato. Ogni driver ha un numero che serve per identificarlo detto **major device**. Se un driver supporta più di un dispositivo, ad esempio due dischi dello stesso tipo, ogni dispositivo ha il proprio **minor device**, un numero che permette di iden-

tificarlo. I due valori insieme (major e minor device) specificano in modo univoco ogni dispositivo I/O. Solo in alcuni casi un driver gestisce dispositivi correlati tra loro: ad esempio, il driver corrispondente a *dev/tty* controlla sia la tastiera sia lo schermo, che vengono spesso pensati come un unico dispositivo: il terminale.

Sebbene non si possa accedere in modo casuale ai file speciali a caratteri, questi spesso hanno bisogno di essere controllati in maniera diversa dai file speciali a blocchi. Si considerino, ad esempio, i caratteri digitati sulla tastiera e visualizzati sullo schermo: quando l'utente commette un errore di battitura e vuole cancellare l'ultimo carattere digitato, egli usa uno dei tasti della tastiera. Alcuni preferiscono posizionarsi all'indietro (usando il *backspace*) altri usano il tasto *DEL*. Allo stesso modo, per cancellare l'intera linea appena digitata, ci sono diverse possibilità: tradizionalmente veniva utilizzato l'*@*, ma visto il notevole utilizzo del carattere per gli indirizzi di e-mail, molti sistemi hanno scelto di usare *CTRL-U*, ed altre possibilità ancora. Per interrompere un programma in esecuzione ci sono alcuni tasti speciali, comunque le preferenze sono molte e diverse tra gli utenti.

Piuttosto che fare una scelta e forzare ciascuno all'uso di certi tasti invece che altri, UNIX permette che tutte queste funzioni e anche altre siano personalizzate dall'utente. Di solito, per decidere queste opzioni viene messa a disposizione una chiamata speciale di sistema che gestisce anche le dimensioni della tabulazione, l'eco del carattere, la conversione del ritorno carrello nell'avanzamento della riga e cose simili. La chiamata di sistema non è permessa su file normali e su file speciali a blocchi.

## Rete

Un altro esempio di I/O è la rete, utilizzata in anteprima dallo UNIX di Berkeley e di cui si parla brevemente di seguito. Il concetto fondamentale nel progetto di Berkeley è il **socket** (presa). I socket sono simili alle caselle postali e alle prese telefoniche, nel senso che permettono agli utenti l'interfaccia con la rete, così come le caselle postali permettono l'interfaccia con il sistema postale e le prese telefoniche consentono di installare i telefoni ed interfacciarsi con il sistema telefonico. La posizione dei socket è mostrata in Figura 10.19.

I socket si possono creare e distruggere dinamicamente: la creazione di un socket restituisce un descrittore di file, che è necessario per stabilire la connessione, leggere i dati, scrivere i dati e rilasciare la connessione.

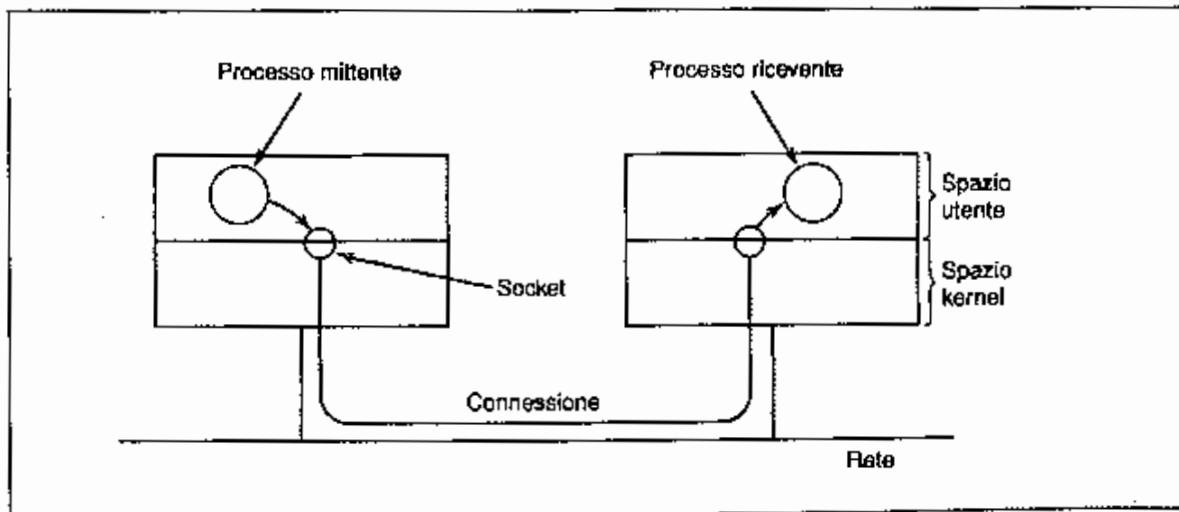


Figura 10.19 L'uso dei socket per la rete.

Ogni socket supporta un particolare tipo di rete, che viene specificata quando si crea il socket. I tipi più comuni sono:

1. Flusso di byte affidabile orientato alla connessione.
2. Flusso a pacchetti affidabile orientato alla connessione.
3. Trasmissione a pacchetti inaffidabile.

Il primo tipo di socket, in effetti, permette a due processi su macchine differenti di stabilire tra loro qualcosa di equivalente ai pipe. I byte sono trasmessi ad una estremità, e vengono emessi all'altra estremità nello stesso ordine in cui sono entrati.

Il secondo tipo è simile al primo tranne per il fatto che mantiene i pacchetti di byte. Se il mittente fa cinque chiamate separate di `write`, ognuna di 512 byte, e il ricevente vuole 2560 byte, con il socket di tipo 1 i 2560 byte saranno restituiti tutti in una sola volta, mentre con il tipo 2 saranno restituiti solo 512 byte, e per ottenere i byte rimanenti sono perciò necessarie quattro chiamate ulteriori. Il terzo tipo di socket si usa per dare all'utente la possibilità di accedere alla rete in modo fisico: questo tipo di socket è utile specialmente per le applicazioni in tempo reale e per quelle situazioni in cui l'utente vuole implementare uno schema particolare di gestione degli errori. I pacchetti possono essere persi o riordinati dalla rete, non vi sono garanzie come nei primi due casi. L'unico vantaggio di questo nodo sono senza dubbio le prestazioni, che talvolta sono preferite all'affidabilità, per esempio per i multimedia, in cui talvolta essere veloci è meglio che essere giusti.

Quando viene creato un socket, uno dei parametri specifica il protocollo da usare. Il protocollo più usato per i flussi a byte affidabili è il TCP, Transmission Control Protocol. Per la trasmissione inaffidabile orientata ai pacchetti di solito si sceglie il protocollo UDP, User Datagram Protocol. Questi due protocolli sono al di sopra del protocollo IP, Internet Protocol. Tutti sono stati realizzati per ARPANET (del Dipartimento della Difesa degli Stati Uniti) ed ora costituiscono la base di Internet. Non c'è un protocollo affidabile a pacchetti altrettanto diffuso.

Prima che un socket possa essere usato per la rete, esso deve essere collegato ad un indirizzo in uno spazio di nomi; il più comune di questi è Internet, che usa interi di 32 bit per la denominazione dei punti terminali nella versione 4 e 128 bit nella versione 6 (la versione 5 era un sistema sperimentale che non è mai stato adottato in grande stile).

Una volta che il socket è stato creato sia sul computer sorgente sia su quello di destinazione, si può stabilire una connessione (per la comunicazione orientata alla connessione). Da un lato, si fa una chiamata di sistema `listen` su di un socket locale, che crea un buffer e resta in attesa dei dati, dall'altro, si fa una chiamata di sistema `connect`, dando come parametri il descrittore del file per un socket locale e l'indirizzo di un socket remoto. Se il partner remoto accetta la chiamata, il sistema stabilisce la connessione tra i due socket.

Una volta stabilita la connessione, il socket funziona come un pipe. Un processo può leggere e scrivere da esso usando il descrittore del file per il socket locale; quando la connessione non è più necessaria, essa può essere chiusa nel solito modo, con la chiamata `close`.

## 10.5.2 Chiamate di sistema per l'I/O in UNIX

Ogni dispositivo di I/O in UNIX, in generale, ha un file speciale ad esso associato. La maggior parte delle operazioni di I/O possono essere fatte semplicemente utilizzando il file relativo, eliminando la necessità di particolari chiamate, ma in certi casi, serve qualcosa di più specifico. Prima di POSIX, molti sistemi UNIX avevano una chiamata di sistema `ioctl` che eseguiva un gran numero di azioni specifiche per il dispositivo sui file speciali. Durante gli anni essa si rivelò decisamente una fonte di confusione. POSIX ripulì la

| Chiamata di funzione                  | Descrizione                       |
|---------------------------------------|-----------------------------------|
| s = cfsetospeed(&termios, velocità)   | Definisce la velocità dell'output |
| s = cfsetspeed(&termios, velocità)    | Definisce la velocità dell'input  |
| s = cgetattrspeed(&termios, velocità) | Recupera la velocità dell'output  |
| s = clgetispeed(&termios, velocità)   | Recupera la velocità dell'input   |
| s = tcsetattr(fd, opz, &termios)      | Definisce gli attributi           |
| s = tcgetattr(fd, &termios)           | Recupera gli attributi            |

Figura 10.20 Le principali chiamate POSIX per la gestione del terminale.

situazione, suddividendo le sue funzionalità fra più funzioni di libreria, intese soprattutto per i terminali. Se siano effettivamente chiamate di sistema distinte, se fanno riferimento alla stessa chiamata, o altro, è lasciato all'implementazione.

Le prime quattro elencate in Figura 10.20 sono usate per definire la velocità dell'input e dell'output del terminale; vengono fornite diverse chiamate per l'input e per l'output perché alcuni modem operano a velocità diverse. Per esempio, vecchi sistemi videotel permettevano a persone che avevano un computer a casa di accedere a basi di dati pubbliche facendo richieste (di solito brevi) dal computer di casa che viaggiavano ad una velocità di 75bit/s, ed ottenendo la risposta (di solito lunga) ad una velocità di 1200bit/s. Questo standard fu adottato quando la velocità di 1200bit/s in entrambe le direzioni era troppo costosa per l'uso domestico, e tale asimmetria sussiste ancora con alcune compagnie telefoniche che offrono un servizio di 1,5 Mbps per il download e 384 Kbps per l'upload, spesso sotto il nome di **ADSL** (Asymmetric Digital Subscriber Line).

Le ultime due chiamate nella lista servono a selezionare ed a rileggere tutti i caratteri speciali usati per cancellare caratteri e linee, per interrompere processi e così via. Inoltre essi abilitano e disabilitano l'eco, gestiscono il controllo del flusso ed altre funzioni. Esistono anche altre chiamate di sistema di I/O, ma sono troppo specializzate per cui non saranno discusse, e comunque, `ioctl` esiste ancora su molti UNIX.

### 10.5.3 Implementazione dell'I/O in UNIX

L'I/O in UNIX viene implementato mediante l'uso di driver di dispositivi, uno per ogni tipo di dispositivo. La funzione dei driver è di isolare il resto del sistema da idiosincrasie dell'hardware, ed inoltre, fornendo interfacce standard tra i driver ed il resto del sistema operativo, la maggior parte del sistema di I/O può essere messo nella parte di kernel indipendente dalla macchina.

Quando un utente accede ad un file speciale, il file system si occupa di determinare i suoi numeri major e minor device e se è un file speciale di blocco o di carattere. Il numero major device viene utilizzato come indice in due strutture di array, `bdevsw` se si tratta di un file speciale di blocco e `cdevsw` se è a carattere. La struttura così recuperata contiene i puntatori alle procedure per accedere al dispositivo, leggerlo, scriverci, e così via. Invece, il numero minor device viene passato come parametro. Quando si aggiunge un nuovo dispositivo per UNIX, un nuovo campo dato viene inserito in una delle due strutture ad array, che specifica le funzioni in grado di eseguire le operazioni da fare sul dispositivo.

Nella struttura `cdevsw`, i campi più importanti per un generico sistema operativo potrebbero essere simili a quelli mostrati in Figura 10.21, dove ogni riga si riferisce ad un singolo dispositivo di I/O (ovvero un singolo driver). Le colonne rappresentano, invece,

| Dispositivo | Apertura | Chiusura  | Lettura  | Scrittura | ioctl     | Altro |
|-------------|----------|-----------|----------|-----------|-----------|-------|
| Null        | null     | null      | null     | null      | null      | ...   |
| Memoria     | null     | null      | mem_read | mem_write | null      | ...   |
| Tastiera    | k_open   | k_close   | k_read   | error     | k_ioctl   | ...   |
| Tty         | tty_open | tty_close | tty_read | tty_write | tty_ioctl | ...   |
| Stampante   | lp_open  | lp_close  | error    | lp_write  | lp_ioctl  | ...   |

Figura 10.21 Alcuni campi tipici di una struttura `cdevsw`.

alcune delle funzioni (ne esistono anche molte altre) che tutti i driver di dispositivi a caratteri devono supportare. Quando viene eseguita una operazione su un file speciale di carattere, il sistema indica dentro l'array `cdevsw` per selezionare il dispositivo associato (riga - driver), quindi chiama la funzione nella colonna per eseguire il lavoro. In conclusione, ogni riga contiene puntatori a funzioni contenute in un certo driver.

Ogni driver viene suddiviso in due parti: la prima metà in alto gira nel contesto del chiamante e si interfaccia con il resto di UNIX; la seconda metà (in basso) gira invece lato kernel ed interagisce con il dispositivo associato. Ai driver è permesso fare particolari richieste al kernel, per esempio per allocare memoria, per la gestione dei timer, per il controllo del DMA ed altre cose. L'insieme di tutte le funzioni che possono essere chiamate è definito precisamente in un documento detto **L'interfaccia Driver-Kernel (Driver-Kernel Interface)**; per imparare come scrivere driver per UNIX si può consultare Egan e Teixeira, 1992.

Il sistema di gestione I/O è diviso in due componenti principali: quella che si occupa della gestione dei file speciali a blocchi e quella per i file speciali a carattere. Vedremo in dettaglio entrambe le componenti.

La finalità principale della parte del sistema dedicata alla gestione dei file speciali a blocchi (ad esempio i dischi) è quella di minimizzare il numero di trasferimenti che devono essere fatti. Per ottenere questo obiettivo, i sistemi UNIX utilizzano una **cache di buffer** (**buffer cache**) posta tra i driver di disco e il file system, come mostrato in Figura 10.22. La cache di buffer non è altro che una tabella presente nel kernel per mantenere migliaia di blocchi utilizzati più recentemente: quando un blocco di memoria viene richiesto al disco per un qualsiasi motivo (i-node, directory, dati), viene fatto un controllo per vedere innanzitutto se esso è presente nella cache di buffer: se è presente, viene recuperato evitando un accesso a disco. Per questo motivo, la cache di buffer migliora notevolmente le prestazioni del sistema.

Se, invece, il blocco non è presente nella cache di buffer, deve essere letto da disco e portato nella cache di buffer, e, da qui, copiato dove necessario. Visto che la cache di buffer può contenere solo un numero fisso di blocchi, inevitabilmente occorre un algoritmo che li gestisca. In particolare, i blocchi nella cache sono collegati tra loro in una lista collegata: ogni volta che si accede ad un blocco, esso viene spostato alla testa della lista. Quando un blocco deve essere rimosso dalla cache per far posto ad un nuovo blocco richiesto, l'algoritmo seleziona il blocco in coda poiché è quello usato meno recentemente.

La cache di buffer lavora per scrivere blocchi analogamente a come fa in fase di lettura: in particolare, quando un programma scrive un blocco, lo scrive sulla cache e non su disco. Solo nel caso in cui la cache è piena ed il buffer serve nuovamente, il blocco viene scritto su disco; e per evitare che blocchi inutilizzati rimangano troppo tempo nella cache di buffer, tutti i blocchi sporchi vengono scritti su disco ogni 30 secondi.

Per decenni, i driver di dispositivo UNIX sono stati collegati staticamente nel kernel,

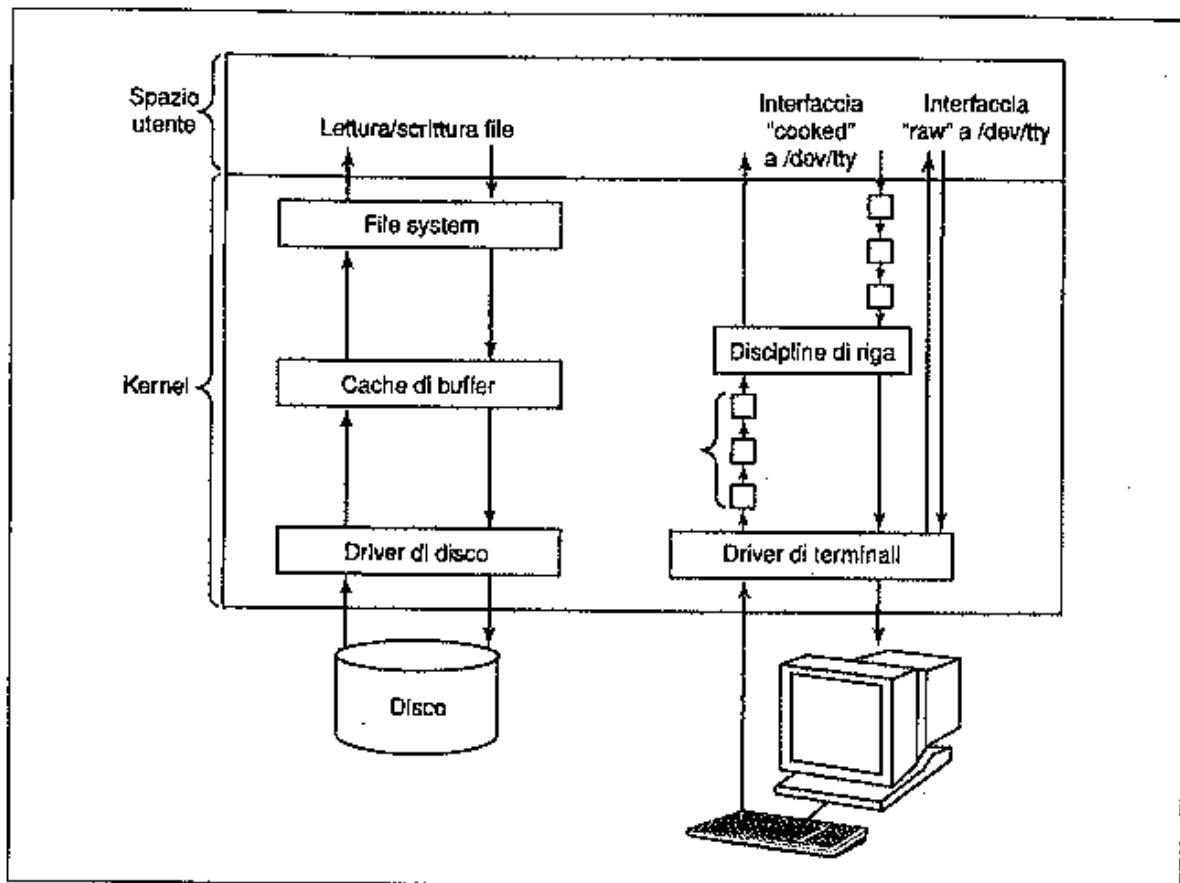


Figura 10.22 Il sistema di I/O nel sistema UNIX BSD.

in modo da essere tutti presenti in memoria quando il sistema veniva avviato, ogni volta. Nel caso dei sistemi su cui UNIX si sviluppò, i micro-computer di dipartimento e workstation di fascia alta dove l'insieme dei dispositivi I/O era piccolo e raramente cambiava, questo schema funzionava alla perfezione. In pratica, un centro di computer costruiva un kernel che conteneva i driver per i dispositivi I/O standard: se l'anno successivo veniva acquistato un nuovo disco, occorreva semplicemente ricollegare il sistema. Nulla di complicato.

Con l'arrivo di Linux, però, su piattaforme PC, tutto cambiò in un momento: il numero di dispositivi disponibili su PC è mille volte maggiore di qualsiasi minicomputer. Inoltre, sebbene gli utenti Linux possiedano il codice sorgente del sistema, o comunque possano procurarselo facilmente, probabilmente la maggior parte di questi avrebbero serie difficoltà ad aggiungere un driver, aggiornando `cdevsw` e `bdevsw`, ricollegare il kernel, ed installarlo come sistema di avvio (per non parlare dell'incubo di cosa fare se il nuovo kernel non parte).

Linux ha cercato di risolvere il problema utilizzando i **moduli caricabili** (loadable module), pezzi di codice che possono essere caricati nel kernel mentre il sistema è in esecuzione. I più comuni sono i driver di dispositivo a blocchi o a carattere, ma ci possono essere anche interi file system, protocolli di rete, strumenti per il controllo delle prestazioni del sistema, e molto altro.

Quando un modulo viene caricato, devono accadere molte cose: primo, il modulo deve essere rilocato al momento, durante il caricamento; secondo, il sistema deve controllare se le risorse di cui ha bisogno (come i livelli di richiesta di interruzione) sono

disponibili e, se si, le deve marcare come in uso. Terzo, ogni vettore di interruzione necessario deve essere impostato; quarto, la tabella di commutazione di driver deve essere aggiornata per poter gestire il nuovo tipo di major device. Infine, si deve permettere al driver di andare in esecuzione per attuare tutte le inizializzazioni necessarie. Una volta completati tutti questi passi, il driver è correttamente installato, esattamente come nel caso di driver caricati staticamente. Oggi esistono altri sistemi UNIX che supportano i moduli caricabili.

## 10.5.4 Stream

Dato che i file speciali di carattere trattano flussi di carattere, e non muovono blocchi di informazione tra memoria e disco, non fanno uso della cache di buffer. Nelle prime versioni di UNIX, ogni driver di dispositivo di carattere faceva tutto il lavoro per il dispositivo collegato. Andando avanti nel tempo, divenne chiaro che molti driver possedevano codice duplicato al loro interno come ad esempio il codice dedicato al buffering, al controllo del flusso, e quello relativo ai protocolli di rete. Vedremo adesso, in dettaglio, le due possibili soluzioni implementate che da un lato strutturano i driver a carattere e dall'altra li modularizzano.

La soluzione del sistema BSD riusa alcune strutture dati presenti in UNIX classico chiamate C-list (C-liste), mostrate come piccoli quadrati in Figura 10.22. Ognuno di questi è un blocco dimensionato fino a 64 caratteri, e possiede anche un contatore ed un puntatore al blocco successivo. Man mano che i caratteri arrivano da terminale o da altri dispositivi a carattere, vengono bufferizzati in catene di blocchi della C-list.

Quando un processo utente legge da `/dev/tty` (cioè lo standard input), i caratteri non vengono direttamente passati dalla C-list al processo, ma passano attraverso un pezzo di codice kernel chiamato **disciplina di riga** (*line discipline*). Tale codice agisce come un filtro, ricevendo in input il flusso di caratteri grezzo da terminale, lo elabora e produce in output un **cooked character stream** (**flusso di caratteri cotto**). In tale flusso, è stata fatta una certa attività di editing sulla riga (sono stati cancellati dei caratteri o delle righe, i ritorno carrello sono stati trasformati in nuova linea), e sono state completate altre azioni speciali. Quindi, il flusso elaborato (cotto) viene passato al processo. Ad ogni modo, se il processo necessita di interagire con ogni carattere, può ricevere la riga in modo grezzo (raw mode) e così evita la disciplina di riga.

La gestione dell'output lavora in modo analogo: i caratteri di tabulazione vengono espansi in spazi, vengono convertiti i caratteri nuova linea in ritorno carrello + nuova linea, vengono aggiunti i caratteri di riempimento dopo il ritorno carrello per terminali meccanici lenti, e così via. Come succede per l'input, l'output può passare attraverso la disciplina di riga (cooked mode) o evitarla (raw mode). Quest'ultima modalità risulta particolarmente utile quando si desidera mandare dati binari ad altri computer da una linea seriale, e per le GUI: in questo caso non si desidera alcuna conversione.

La soluzione di System V, invece, è più generale (mostrata in Figura 10.23) ed utilizza i **flussi** (**streams**), inventati da Dennis Ritchie (si noti che anche System V possiede una cache di buffer per i file speciali a blocchi, che è quasi uguale a quella di BSD, che qui non viene mostrata). L'idea di base di un flusso è quella di poter connettere un processo utente al driver e di inserire i moduli di elaborazione nello stream dinamicamente, in fase di esecuzione. In un certo senso, lo stream nel kernel può essere analogo alle pipeline nello spazio utente.

Uno stream possiede sempre una testa e connette un driver che si trova in fondo: possono essere inseriti tutti i moduli necessari; l'elaborazione può essere bidirezionale, in modo che ogni modulo può avere una sezione di lettura (dal driver) ed una di scrittura

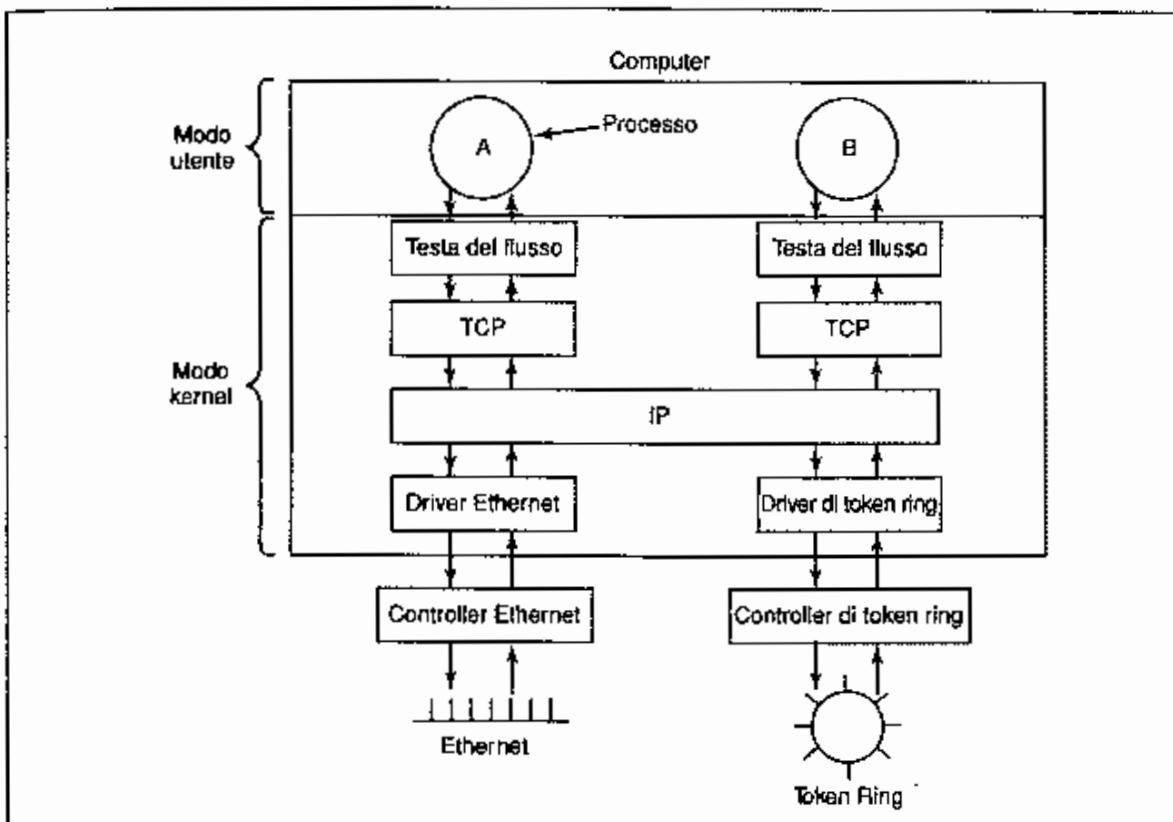


Figura 10.23 Un esempio di stream in System V.

(al driver). Quando un processo utente scrive sullo stream, il codice in testa interpreta la chiamata di sistema e compatta i dati nei buffer che vengono poi passati da modulo a modulo, verso il basso, attuando tutte le trasformazioni che devono essere fatte. Ogni modulo mantiene una coda di lettura ed una coda di scrittura, in modo da elaborare i buffer sempre in ordine corretto; inoltre i moduli posseggono delle interfacce ben definite in modo da poter agganciare moduli non collegati.

L'esempio in Figura 10.23 mostra come gli stream sono usati quando si sta utilizzando un protocollo TCP Internet in due tipi diversi di rete locale, Ethernet e Token ring. In particolare, l'esempio illustra una caratteristica ulteriore degli stream - il multiplexing: un modulo multiplexing può prendere uno stream e dividerlo in più stream e viceversa (il modulo IP in questa configurazione attua entrambe le operazioni).

## 10.6 Il file system di UNIX

La parte di un sistema maggiormente in evidenza, anche in UNIX, è il file system. Nelle sezioni che seguono cercheremo di esaminare le idee di base del file system di UNIX, le chiamate di sistema e l'implementazione dei file. Alcune di queste idee derivano da MULTICS e molte altre sono state copiate da MS-DOS, Windows ed altri sistemi, ma alcune sono uniche di UNIX. Il progetto UNIX è interessante soprattutto come prova che *Piccolo è Bello (Small is beautiful)*. Con un meccanismo minimo, ed un numero di chiamate di sistema irrisorio, UNIX è in grado di fornire un file system potente ed elegante.

## 10.6.1 Concetti di base

Un file UNIX è una sequenza di 0 o più byte che contengono informazioni di qualsiasi genere; inoltre non viene fatta alcuna distinzione tra file ASCII, file binari o qualsiasi altro tipo di file. Il significato dei bit in un file dipende del tutto dal proprietario, al sistema non interessa. All'inizio i nomi dei file dovevano essere non più lunghi di 14 caratteri, ma lo UNIX di Berkeley incrementò questo limite a 255 caratteri, cosa che fu adottata anche in System V ed in altre versioni. Sono permessi tutti i caratteri ASCII, tranne NUL, nei nomi di file, quindi un nome di file formato da tre caratteri "ritorno carrello" è un nome di file legale (ma non uno particolarmente conveniente).

Per convenzione, molti programmi si aspettano nomi di file costituiti da un nome di base e da un'estensione, separata da un punto (dot, che è considerato un carattere). Così *prog.c* è un programma C, *prog.f90* è un programma FORTRAN 90 e *prog.o* è di solito un file oggetto (output del compilatore). Queste convenzioni non sono imposte dal sistema operativo ma alcuni compilatori ed altri programmi le attendono. Le estensioni possono essere di qualsiasi lunghezza, ed i file possono avere estensioni multiple, ad esempio *prog.java.Z* (presumibilmente un programma java compresso).

I file si possono raggruppare insieme per comodità nelle directory. Le directory sono memorizzate come file e sono sovente trattate come file; esse possono contenere sottodirectory, dando luogo così a un file system gerarchico. La directory radice è detta / e di solito contiene diverse sottodirectory. Il carattere / è usato anche per separare i nomi delle directory, per cui /usr/ast/x indica che il file x è nella directory ast, la quale appartiene alla directory usr, a sua volta contenuta nella directory radice.

La Figura 10.24 mostra alcune delle principali directory del file system di UNIX, solitamente poste vicino alla radice.

Ci sono due modi per specificare i nomi dei file in UNIX, sia alla shell sia all'apertura del file da programma: il primo modo usa il cammino assoluto (*absolute path*), cioè il cammino che si deve percorrere dalla directory radice per raggiungere il file. Come esempio di cammino assoluto si consideri /usr/ast/libro/mos2/cap-10; in questo modo si dice al sistema di cercare, nella directory radice, una directory chiamata *usr*, in questa cercare quella chiamata *ast* che contiene la directory *libro* poi nella directory *mos2* dove infine si trova il file *cap-10*.

I cammini assoluti sono spesso troppo lunghi e poco convenienti, perciò, UNIX permette agli utenti ed ai processi di dichiarare la directory, che normalmente usano per lavorare, come directory di lavoro (*working directory*). I pathname si possono specificare anche in relazione alla directory di lavoro, un nome di questo tipo è chiamato cammino relativo (*relative path*). Per esempio, se /usr/ast/libro/mos2 è la directory di lavoro, allora il comando shell

| Directory | Contenuto                            |
|-----------|--------------------------------------|
| bin       | Programmi binari eseguibili          |
| dev       | File speciali per dispositivi di I/O |
| etc       | File di sistema vari                 |
| lib       | Librerie                             |
| usr       | Directory di utente                  |

Figura 10.24 Alcune directory importanti per sistemi UNIX.

`cp cap-10 backup1`

ha lo stesso effetto del seguente comando più lungo

`cp /usr/ast/libro/mos2/cap-10 /usr/ast/libro/mos2/backup1`

Succede spesso che un utente abbia bisogno di fare riferimento a un file appartenente ad un altro utente, o che si trova da un'altra parte dell'albero dei file. Per esempio, se due utenti stanno condividendo un file, questo sarà sistemato in una directory di uno dei due, per cui l'altro dovrà usare un pathname assoluto per far riferimento a quel file, oppure cambiare la propria directory di lavoro. Se il pathname assoluto è molto lungo, digitarlo tutto più volte può innervosire l'utente, per cui UNIX offre una soluzione a questo problema permettendo all'utente di creare un nuovo elemento di directory che punta al file esistente. Questo tipo di elemento è detto **link** (collegamento).

Come esempio, consideriamo la situazione di Figura 10.25(a). Fred e Lisa stanno lavorando insieme ad un progetto ed ognuno di essi ha spesso bisogno di accedere al file dell'altro. Se Fred ha come directory di lavoro `/usr/fred`, egli può fare riferimento al file `x` nella directory di Lisa con `/usr/lisa/x`. In modo alternativo, Fred può creare un nuovo elemento nella sua directory come mostrato nella Figura 10.25(b) e quindi usare semplicemente `x` per sottintendere `/usr/lisa/x`.

Nell'esempio appena discusso, si è detto che prima del collegamento, l'unico modo per Fred di far riferimento al file `x` di Lisa era l'uso del suo pathname assoluto, ma questo non è del tutto vero. Quando viene creata una directory, in modo automatico in essa vengono messi due elementi: `."` e `..`: il primo fa riferimento alla directory stessa, il secondo fa riferimento alla directory genitore, per cui da `/usr/fred` un altro path al file `x` di Lisa è `../lisa/x`.

Oltre ai file standard, UNIX supporta anche i file speciali a blocchi ed a carattere; questi ultimi sono utilizzati per modellare i dispositivi seriali di I/O come tastiera e stampanti. L'apertura e la lettura da `/dev/tty` legge da tastiera; l'apertura e la scrittura su `/dev/lp` scrive sulla stampante. I file speciali a blocchi, spesso con nomi tipo `/dev/hd1`, possono essere usati per leggere e scrivere su partizioni disco grezze, senza considerare il file system. Quindi una ricerca del byte `k` seguita da una lettura dei dati, leggerà il `k`-esimo

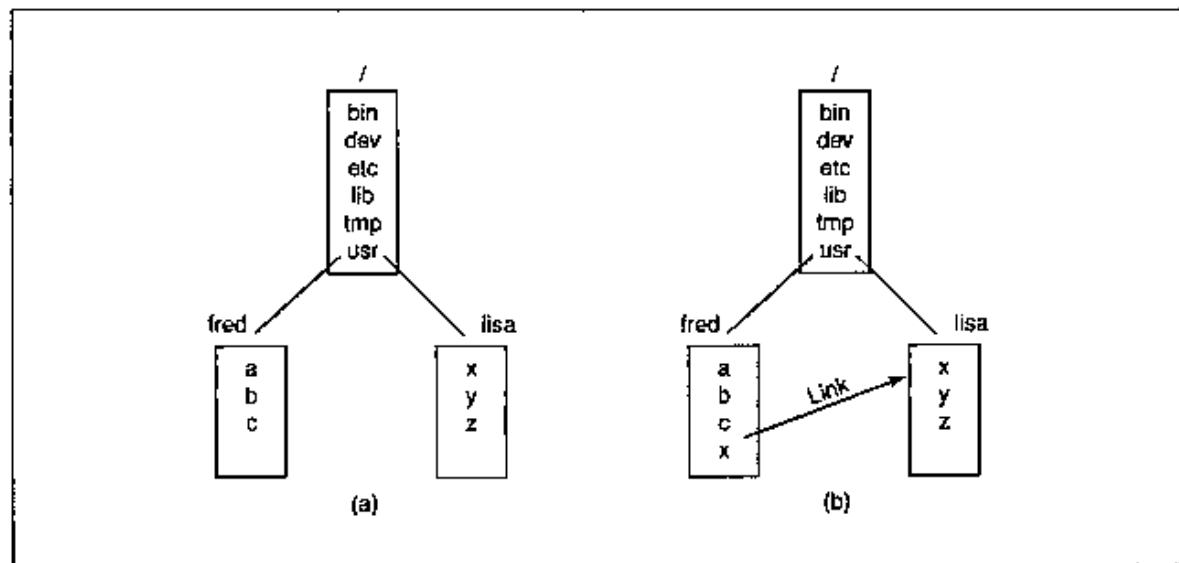


Figura 10.25 (a) Prima del collegamento. (b) Dopo il collegamento.

byte sulla partizione corrispondente, senza tener conto degli i-node e della struttura del file system. I dispositivi a blocchi sono usati in modo grezzo anche per la paginazione e per lo swapping da programmi, ad esempio, che costruiscono il file system (*mkfs*) e da programmi che riparano i problemi del file system (*fsck*).

Molti computer hanno due o più dischi. Nei grossi calcolatori delle banche, per esempio, è spesso necessario avere 100 o più dischi in una sala macchina per mantenere l'enorme base di dati utilizzata. Anche i personal computer hanno di solito almeno due dischi - un disco rigido e un drive per i floppy disk. Quando ci sono più periferiche di dischi si ha il problema di come gestirli.

Una soluzione è mettere un file system autocontenuto su ognuno di essi e quindi gestirli in modo separato. Si consideri, per esempio, la situazione mostrata in Figura 10.26(a), in cui si ha un disco rigido chiamato C: ed un dischetto chiamato A:. Ognuno di essi ha la sua directory radice ed i suoi file. In questo modo, l'utente deve specificare sia la periferica che il file. Ad esempio, per copiare il file *x* sulla directory *d* (prendendo da C: per difetto) si digiterebbe:

```
cp A:/x /a/d/x
```

Questo tipo di approccio si trova sui sistemi MS-DOS e VMS.

La soluzione UNIX permette di montare un disco nell'albero dei file di un altro disco. Nell'esempio dato, si potrebbe montare il dischetto sulla directory /*b*, ottenendo il sistema di Figura 10.26 (b). Ora l'utente ha a disposizione un solo albero di file e non deve più preoccuparsi di dove risiede il file e su quale periferica. Il precedente comando di copia diventa

```
cp /b/x /a/d/x
```

esattamente come sarebbe stato se tutto si fosse trovato sul disco rigido.

Un'altra proprietà interessante del file system di UNIX è il bloccaggio (locking). In alcune applicazioni, due o più processi possono richiedere l'utilizzo dello stesso file nello stesso momento, il che può portare a una corsa critica. Una soluzione è quella di programmare l'applicazione con delle regioni critiche, ma se i processi appartengono ad utenti indipendenti che non si conoscono, questo tipo di coordinazione generalmente è poco conveniente.

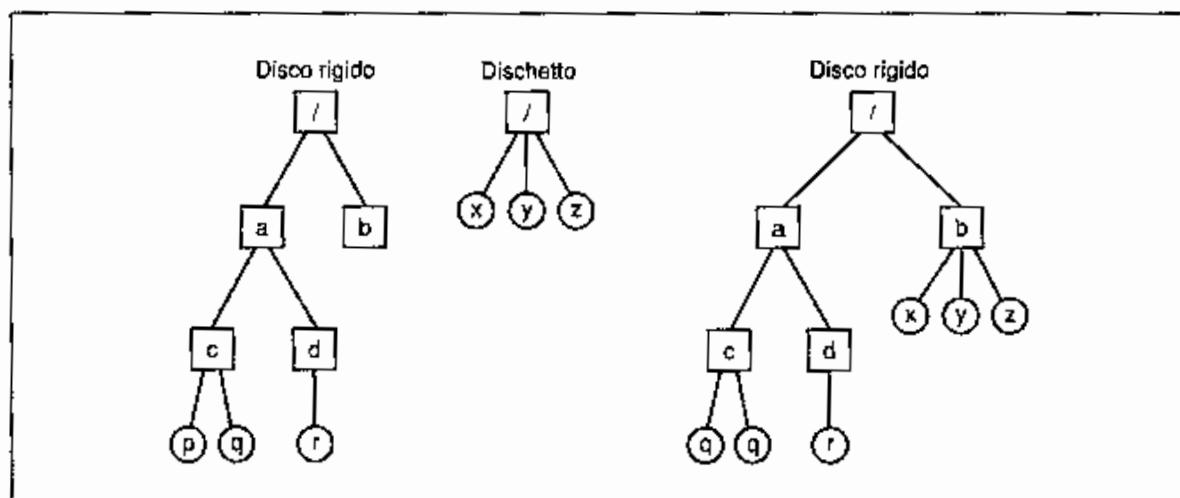


Figura 10.26 (a) File system separati. (b) Dopo il mount.

Si consideri, ad esempio, una base di dati formata da molti file in una o più directory visitate da utenti che non sono in relazione: certamente si può associare un semaforo ad ogni directory o file ed avere la mutua esclusione, purché i processi facciano un'operazione down sul semaforo appropriato prima dell'accesso ai dati. Lo svantaggio, però, è che un'intera directory o file è resa inaccessibile, anche nel caso sia necessario un solo record.

Per questa ragione, POSIX fornisce un meccanismo ben strutturato e flessibile che permette ai processi di bloccare un singolo byte o un intero file con una operazione unica ed indivisibile. Il meccanismo di lock richiede che il chiamante specifichi il file che deve essere bloccato, il byte di partenza e il numero di byte; se l'operazione ha successo, il sistema crea una tabella di elementi in cui è scritto che i byte in questione (per esempio un record di una base di dati) sono bloccati.

Sono forniti due tipi di lock, **condivisi (shared)** ed **esclusivi (exclusive)**. Se una porzione di un file contiene già un lock condiviso, è possibile mettere un secondo lock di questo tipo in esso, mentre il tentativo di mettere un lock esclusivo fallirà. Se una porzione di file contiene un lock esclusivo, tutti i tentativi di bloccare qualunque parte di quella porzione falliranno finché il lock non sarà tolto. Per poter bloccare una porzione di file, ogni byte della regione deve essere a disposizione.

Quando vuole mettere un lock, un processo deve specificare se vuole interrompersi o no nel caso che il lock non si possa effettuare. In caso affermativo, quando il lock esistente è stato rimosso il processo viene sbloccato ed effettua il lock da lui richiesto. Altrimenti, la chiamata di sistema termina immediatamente con il codice di stato che indica se il lock ha avuto successo o no.

Le regioni bloccate possono sovrapporsi: nella Figura 10.27(a) si vede che il processo A ha effettuato un lock condiviso dal byte 4 al byte 7 di un certo file. In un secondo momento, il processo B ha effettuato un lock condiviso dal byte 6 al byte 9, come mostrato in Figura 10.27 (b). Infine, C effettua un lock dal byte 2 al byte 11; se tutti questi lock sono condivisi, possono coesistere. Si consideri ora cosa accade se un processo prova ad

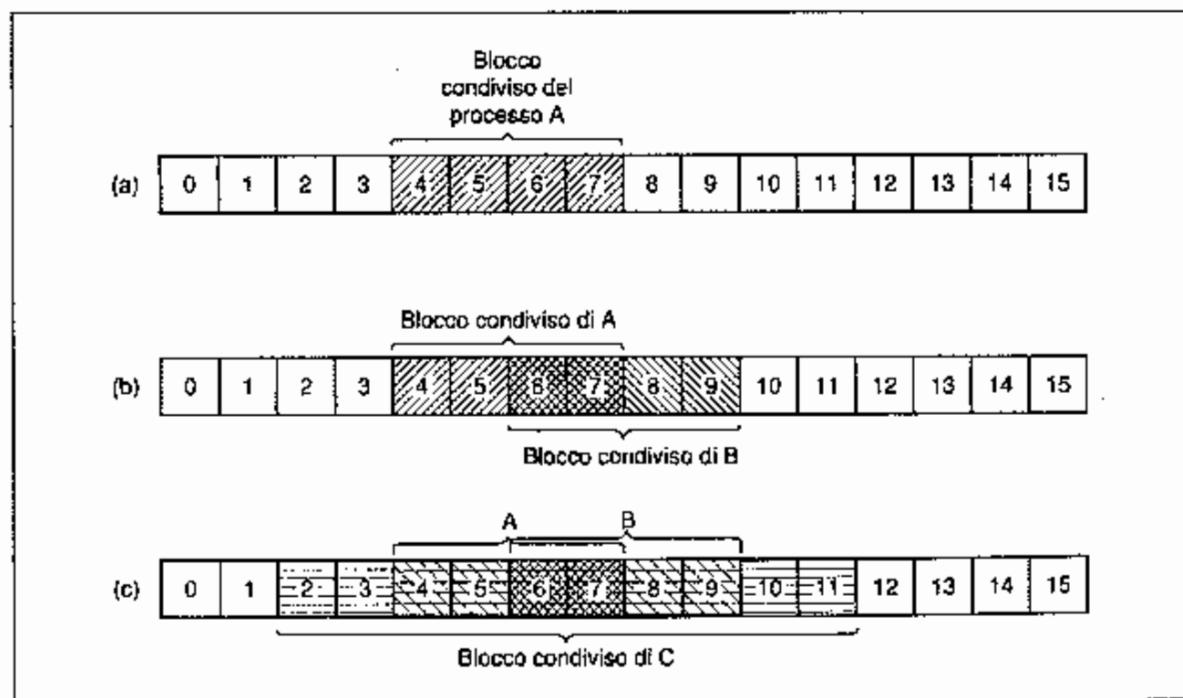


Figura 10.27 [a] Un file con un lock. [b] L'aggiunta di un secondo lock. [c] Un terzo lock.

ottenere un lock esclusivo sul byte 9 di Figura 10.27(c), con la richiesta di interrompersi se il lock fallisce. Poiché due lock precedenti sono già presenti su questo byte, il chiamante sarà interrotto e resterà bloccato finché sia B sia C non rilasceranno il loro lock.

## 10.6.2 Chiamate di sistema per i file in UNIX

Molte chiamate di sistema si riferiscono ai file ed al file system; si esamineranno prima le chiamate di sistema che operano sui file singoli, in un secondo momento si esamineranno quelle che coinvolgono le directory o il file system. Per creare un nuovo file si usa la chiamata `creat`. (Quando a Ken Thompson fu chiesto una volta cosa avrebbe fatto in modo diverso se avesse avuto la possibilità di reinventare UNIX, egli rispose che avrebbe cambiato la chiamata `creat` in `create`). I parametri forniscono il nome del file e la modalità di protezione. Per cui

```
fd = creat("abc", mode);
```

crea un file chiamato *abc* con i bit di protezione presi da *mode*. Tali bit stabiliscono che utenti possono accedere al file, e come; li descriveremo più avanti.

`creat` non solo crea un nuovo file, ma lo apre anche per la scrittura, e se ha successo, permette di usarlo in chiamate di sistema successive, tramite il valore restituito, un piccolo intero positivo, detto **descrittore del file**, che nell'esempio si chiama *fd*. Se si fa una `creat` su un file già esistente, quel file è troncato a lunghezza 0, eliminando il suo contenuto.

Continuiamo adesso ad analizzare le principali chiamate di sistema elencate in Figura 10.28. Per leggere o scrivere su un file già esistente, questo deve essere prima aperto usando una `open`, che specifica il nome del file da aprire e se esso si deve aprire per la sola lettura, per la scrittura o per entrambe le modalità. Si possono, ad ogni modo, specificare diversi tipi di opzioni, e proprio come per `creat`, il descrittore di file restituito può poi essere usato in varie operazioni di lettura e scrittura. Successivamente il file può essere chiuso con una `close`, che rende il descrittore del file disponibile ad un possibile riutilizzo in una successiva `creat` o `open`. Entrambe le chiamate restituiscono sempre il più basso descrittore di file non in uso.

| Chiamata di sistema                                | Descrizione                                    |
|----------------------------------------------------|------------------------------------------------|
| <code>fd = creat(nome, modo)</code>                | Un modo per creare un nuovo file               |
| <code>fd = open(file, come, ...)</code>            | Apre un file per lettura, scrittura o entrambe |
| <code>s = close(fd)</code>                         | Chiude un file aperto                          |
| <code>n = read(fd, buffer, nbyte)</code>           | Legge dati da un file a un buffer              |
| <code>n = write(fd, buffer, nbyte)</code>          | Scrive dati da un buffer a un file             |
| <code>position = lseek(fd, offset, da_dove)</code> | Muove il puntatore del file                    |
| <code>s = stat(nome, &amp;buf)</code>              | Recupera le informazioni dello stato del file  |
| <code>s = fstat(fd, &amp;buf)</code>               | Recupera le informazioni dello stato del file  |
| <code>s = pipe(&amp;fd[0])</code>                  | Crea una pipe                                  |
| <code>s = fcntl(fd, cmd, ...)</code>               | Locking di file ed altre operazioni            |

**Figura 10.28** Alcune chiamate di sistema relative ai file. Il codice restituito *s* è -1 in caso di errore; *fd* è il descrittore di file; *position* è l'offset del file. Gli altri parametri sono autoesplicativi.

Quando un programma inizia ad essere eseguito i descrittori di file 0, 1 e 2 sono già in uso per lo standard input, output e lo standard error. In questo modo, un filtro, come ad esempio il programma *sort*, legge l'input del descrittore di file 0 e lo butta nel descrittore di file 1 senza chiedersi che file siano. Tutto ciò funziona perfettamente perché la shell imposta (ridirige) questi file prima che il programma inizi.

Tra le possibili chiamate le più usate, sono indubbiamente *read* e *write*. Entrambe posseggono tre parametri: un descrittore di file (che dice quale file aperto leggere o scrivere), un indirizzo di buffer (che descrive dove mettere i dati o dove recuperarli) ed un contatore (che dice quanti byte devono essere trasferiti), e basta. È uno schema molto semplice ed una tipica chiamata ha la forma

```
n = read(fd, buffer, nbytes);
```

Sebbene molti programmi leggano e scrivano file in modo sequenziale, per alcuni tipi di applicazioni i programmi devono accedere a parti del file in modo casuale. Associato ad ogni file esiste un puntatore che indica la posizione corrente nel file, che durante la lettura (scrittura) in modo sequenziale, punta normalmente al byte successivo che deve essere letto (scritto). Se il puntatore è, ad esempio 4096, prima di leggere i successivi 1024 byte, si muoverà automaticamente a 5120 dopo una chiamata con successo di *read*. La chiamata *lseek* cambia il valore del puntatore di posizione, in modo che le chiamate successive a *read* o a *write* possano iniziare in qualsiasi parte del file o anche oltre la fine del file stesso.

*lseek* ha tre parametri: il primo è il descrittore del file; il secondo è una posizione del file; il terzo dice se la posizione è relativa all'inizio del file, alla posizione corrente o alla fine del file. Il valore restituito da *lseek* è la posizione assoluta nel file dopo che il puntatore del file è stato cambiato. In modo abbastanza ironico, *lseek* è l'unica chiamata di sistema che non può far cercare nulla su disco, visto che semplicemente aggiorna la posizione corrente del file (un numero in memoria).

Per ciascun file, UNIX mantiene traccia del tipo del file (regolare, directory, speciale), della dimensione, dell'ora dell'ultima modifica ed altre informazioni. I programmi possono richiedere queste informazioni usando la chiamata *stat*, il cui primo parametro è il nome del file, il secondo è un puntatore ad una struttura dove si deve inserire l'informazione richiesta. I campi nella struttura sono mostrati in Figura 10.29; la chiamata *fstat* opera allo stesso modo di *stat* eccetto per il fatto che opera su di un file aperto (il cui nome potrebbe non essere conosciuto) piuttosto che su un pathname.

|                                               |
|-----------------------------------------------|
| Dispositivo su cui sta il file                |
| Numero di i-node [quale file sul dispositivo] |
| Modo del file (comprese le info di sicurezza) |
| Numero di collegamenti al file                |
| Identità del proprietario del file            |
| Gruppo a cui il file appartiene               |
| Dimensione del file (in byte)                 |
| Data di creazione                             |
| Ultimo accesso                                |
| Ultima modifica                               |

Figura 10.29 I campi restituiti dalla chiamata *stat*.

La chiamata di sistema `pipe` viene utilizzata per la creazione di una pipeline di shell: crea uno pseudofile che bufferizza i dati tra le componenti della pipe e restituisce i descrittori di file per leggere o scrivere nel buffer. In una pipeline tipo

```
sort <in|head -30
```

il descrittore di file 1 (standard output) nel processo che sta eseguendo `sort` scriverà nella pipe, ed il descrittore di file 0 (standard input) nel processo che sta eseguendo `head` leggerà dalla pipe. In questo modo, `sort` leggerà dal descrittore di file 0 (impostato per il file `in`) e scriverà nel descrittore di file 1 (pipe) senza preoccuparsi se il tutto verrà ridiretto o meno; se non fossero stati ridiretti, `sort` avrebbe automaticamente letto dalla tastiera e scritto sullo schermo (i dispositivi per difetto). Similmente, quando `head` legge dal descrittore 0 legge di fatto i dati messi da `sort` nella pipe senza accorgersi che la pipe è in uso. È questo un semplice esempio in cui un semplice concetto (ridirezione) con una semplice implementazione (descrittori di file 0 e 1) permette di ottenere uno strumento molto potente (la connessione in modo arbitrario di programmi senza doverli modificare).

L'ultima chiamata di sistema in Figura 10.28 è `fcntl` che viene usata per bloccare e sbloccare dei file, e compiere alcune operazioni specifiche dei file.

Ora si esamineranno alcune chiamate di sistema che riguardano più directory o tutto il file system, piuttosto che un file specifico (si veda Figura 10.30). Le directory sono create e distrutte usando rispettivamente le chiamate `mkdir` e `rmdir`; una directory può essere rimossa solo se è vuota.

Come si è visto in Figura 10.25, il collegamento ad un file crea un nuovo elemento della directory che punta ad un file esistente. La chiamata di sistema `link` crea l'aggancio; i parametri specificano rispettivamente il nome originale e quello nuovo. Gli elementi della directory sono rimossi con la chiamata `unlink`; quando viene rimosso l'ultimo aggancio ad un file, questo è automaticamente cancellato. Se un file non ha mai avuto un aggancio, basta una `unlink` per cancellarlo.

La directory di lavoro è modificata dalla chiamata di sistema `chdir` e ciò ha l'effetto di modificare l'interpretazione dei pathnames relativi.

Le ultime quattro chiamate in Figura 10.30 sono dedicate alla lettura di directory, che possono essere aperte, chiuse, e lette analogamente a come si fa per i file ordinari. Ogni chiamata di `readdir` restituisce una directory in un formato fissato; non esiste modo per

| Chiamata di sistema                           | Descrizione                                     |
|-----------------------------------------------|-------------------------------------------------|
| <code>s = mkdir(path, modo)</code>            | Crea una nuova directory                        |
| <code>s = rmdir(path)</code>                  | Rimuove una directory                           |
| <code>s = link(pathvecchio, pathnuovo)</code> | Crea un collegamento ad una directory esistente |
| <code>s = unlink(path)</code>                 | Elimina un collegamento ad un file              |
| <code>s = chdir(path)</code>                  | Cambia la directory di lavoro                   |
| <code>dir = opendir(path)</code>              | Apre una directory in lettura                   |
| <code>s = closedir(dir)</code>                | Chiude una directory                            |
| <code>dirent = readdir(dir)</code>            | Legge un campo dato di directory                |
| <code>rewinddir(dir)</code>                   | Ricarica una directory in modo da leggerla      |

Figura 10.30 Alcune chiamate di sistema legate alle directory. Il codice restituito s è -1 in caso di errore, `dir` identifica lo stream di directory, e `dirent` è il campo dato directory. Gli altri parametri dovrebbero essere autoesplcativi.

gli utenti di scrivere in una directory (questo per mantenere integrità nel file system). I file possono invece essere aggiunti ad una directory usando `creat` o `link` e rimossi utilizzando `unlink`. Non esiste, infine, un modo per cercare un file in una directory, ma `rewinddir` permette di rileggere dall'inizio una directory aperta.

### 10.6.3 Implementazione del file system di UNIX

In questo paragrafo si descriverà l'implementazione del file system tradizionale di UNIX, in un secondo momento si parlerà di quello di Berkeley, e ce ne sono altri. Tutti i sistemi UNIX possono supportare partizioni multiple di disco, ognuna con un diverso file system.

Una tipica partizione UNIX contiene un file system organizzato come illustra la Figura 10.31. Il blocco 0 non è usato da UNIX e spesso contiene il codice per inizializzare il computer; il blocco 1 è il **superblocco** (**superblock**), che contiene informazioni critiche sulla struttura del file system, come il numero di i-node, il numero di blocchi del disco e l'inizio della lista dei blocchi liberi (tipicamente poche centinaia di elementi). La distribuzione del superblocco renderà il file system non leggibile.

Di seguito al superblocco ci sono gli **i-node** (abbreviazione di index-node, mai chiamati in questo modo, anzi, alcuni particolarmente pigri buttano il trattino e li chiamano **inode**) che sono numerati da 1 ad un numero massimo. Ogni i-node è lungo 64 byte e descrive esattamente un file; esso contiene delle informazioni di contabilità (le stesse restituite da `stat` che di fatto utilizza proprio l'i-node del file) e tutte le informazioni per localizzare tutti i blocchi del disco su cui si trovano i dati del file.

Dopo gli i-node ci sono i blocchi dei dati; qui sono memorizzati tutti i file e le directory. Se un file o una directory sono formati da più di un blocco, non occorre che essi siano contigui sul disco; infatti i blocchi di un grosso file sono sparsi sul disco ed è a questa dispersione che ha prestato maggior attenzione il progetto del sistema realizzato a Berkeley, per tentare di ridurla.

Nel file system tradizionale (V7) una directory è formata da un insieme non ordinato di elementi di 16 byte; ogni elemento contiene un nome di file (fino a 14 caratteri qualsiasi) ed il numero dell'i-node del file come mostrato in Figura 6.37. Per aprire un file nella directory di lavoro, il sistema legge la directory, confronta il nome in ingresso con ogni elemento finché non trova un nome corrispondente o scopre che non esiste.

Se il file è presente, estrae il numero di i-node e lo usa come indice nella tabella di i-node (su disco) per localizzare l'i-node corrispondente e portarlo in memoria. L'i-node è messo nella tabella di i-node che è una struttura dati del kernel contenente tutti gli i-node dei file e delle directory aperte. Il formato di un i-node può variare leggermente da versione a versione di UNIX, ma come minimo deve possedere i campi che vengono utilizzati da `stat` (si veda la Figura 10.29) perché tale chiamata possa funzionare. In Figura

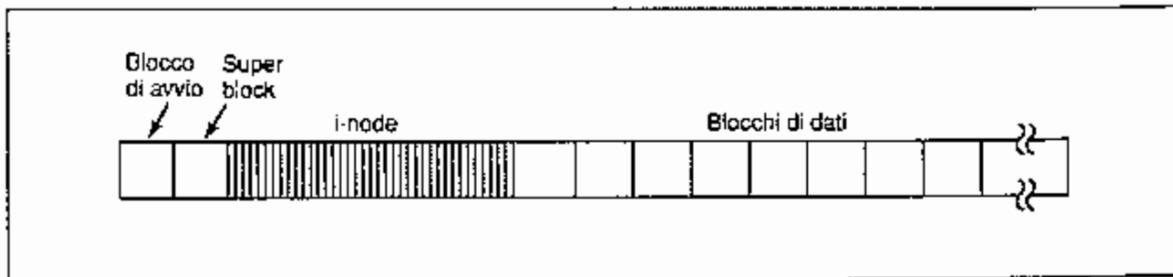


Figura 10.31 Il layout del disco in un sistema UNIX standard.

| Campo  | Byte | Descrizione                                                             |
|--------|------|-------------------------------------------------------------------------|
| Mode   | 2    | Tipo di file, bit di protezione, setuid e setgid                        |
| Nlinks | 2    | Numero di campi dato di directory che puntano a questo i-node           |
| Uid    | 2    | UID del proprietario di file                                            |
| Gid    | 2    | GID del proprietario di file                                            |
| Size   | 4    | Dimensione del file in byte                                             |
| Addr   | 39   | Indirizzo dei primi 10 blocchi disco, quindi 3 blocchi indiretti        |
| Gen    | 1    | Numero di generazione (aumentato ogni volta che l'i-node viene riusato) |
| Atime  | 4    | Data ultimo accesso                                                     |
| Mtime  | 4    | Data ultima modifica                                                    |
| Ctime  | 4    | Data dell'ultimo cambiamento di i-node                                  |

Figura 10.32 La struttura di un i-node in System V.

ra 10.32 si può osservare il formato utilizzato da tutte le versioni AT&T dalla versione 7 a System V.

Seguire il pathname assoluto /usr/ast/file è una operazione più complicata: il sistema inizia con il localizzare la directory radice, che usa sempre l'i-node 2 (infatti l'i-node 1 è riservato per la gestione dei blocchi danneggiati), poi cerca la stringa "usr" nella directory radice per conoscere il numero di i-node di /usr, che quindi ricupera, e da cui estrae i blocchi della directory in esame. A questo punto /usr può essere letto e in essa il file system cercherà la stringa "ast". Trovata la corrispondenza, avrà a disposizione il relativo numero di i-node, che verrà letto, e permetterà di andare a leggere i blocchi della directory /usr/ast. Finalmente si cercherà ora la stringa corrispondente a "file" ed il suo relativo numero di i-node. Da tutto ciò si deduce che l'utilizzo di un pathname relativo non è solo più conveniente per l'utente, ma costa meno lavoro al sistema.

Ora vediamo come il sistema legge un file. Si ricorda che una tipica chiamata di sistema `read` è del tipo:

```
n = read (fd, buffer, nbytes);
```

Quando il kernel prende il controllo, deve partire con questi tre parametri e con l'informazione che si trova nella struttura utente del chiamante. Uno degli oggetti della struttura utente è la tabella dei descrittori dei file, che è indirizzata dai descrittori di file e contiene un elemento per ogni file aperto (il numero massimo di solito è 20).

Si parte allora con questo descrittore di file e si termina con il corrispondente i-node. Si consideri un possibile progetto: mettere un puntatore all'i-node nella tabella dei descrittori di file. Tale metodo, sebbene sia semplice, sfortunatamente non funziona, infatti presenta il seguente problema: associata ad ogni descrittore di file esiste una posizione del file che informa da quale byte partirà la prossima lettura (scrittura). Dove sta questa informazione? Una possibilità potrebbe essere di metterlo nella tabella dell'i-node, ma questo approccio fallisce se due processi, che non sono in alcuna relazione tra loro, decidono di aprire lo stesso file nello stesso momento perché ognuno ha la sua propria posizione.

Una seconda possibilità è quella di mettere la posizione del file nella tabella dei descrittori di file; in questo modo ogni processo che apre un file ha la sua propria posizione di file. Sfortunatamente anche questo schema non funziona, e la ragione di ciò è più sottile ed ha a che fare con i file condivisi di UNIX. Si consideri uno shell script s, forma-

to da due comandi  $p1$  e  $p2$  che devono essere eseguiti nell'ordine. Se lo shell script viene chiamato da una linea di comando del tipo

`s >x`

ci si aspetta che  $p1$  scriva il suo output su  $x$  ed anche  $p2$  lo scriva su  $x$ , iniziando da dove ha terminato  $p1$ .

Quando la shell fa una fork per  $p1$ ,  $x$  è inizialmente vuoto, per cui  $p1$  inizia a scrivere dalla posizione 0, ma quando esso termina, è necessario qualche meccanismo che assicuri che la posizione di partenza di  $p2$  non sia 0 bensì la fine del file  $x$  (cosa che accadrebbe se la posizione del file fosse mantenuta nella tabella del descrittore del file).

La Figura 10.33 mostra come si raggiunge questo obiettivo; basta, infatti, introdurre una nuova tabella, la tabella dei descrittori dei file aperti, tra quella dei descrittori di file e quella di i-node, ed inserivi la posizione del file (ed il bit di lettura/scrittura). Nella figura, il genitore è la shell ed il figlio è prima  $p1$  e poi  $p2$ ; quando la shell fa la fork di  $p1$ , la sua struttura utente (compresa la tabella dei descrittori dei file) è una copia esatta di quella della shell, per cui entrambi i file puntano allo stesso elemento della tabella dei descrittori dei file aperti. Quando  $p1$  termina, il descrittore dei file della shell sta ancora puntando al descrittore dei file aperti che contiene la posizione del file di  $p1$ . Al momento in cui la shell fa una fork di  $p2$ , il nuovo figlio eredita automaticamente la posizione del file e non è necessario che né esso né tanto meno la shell lo conoscano.

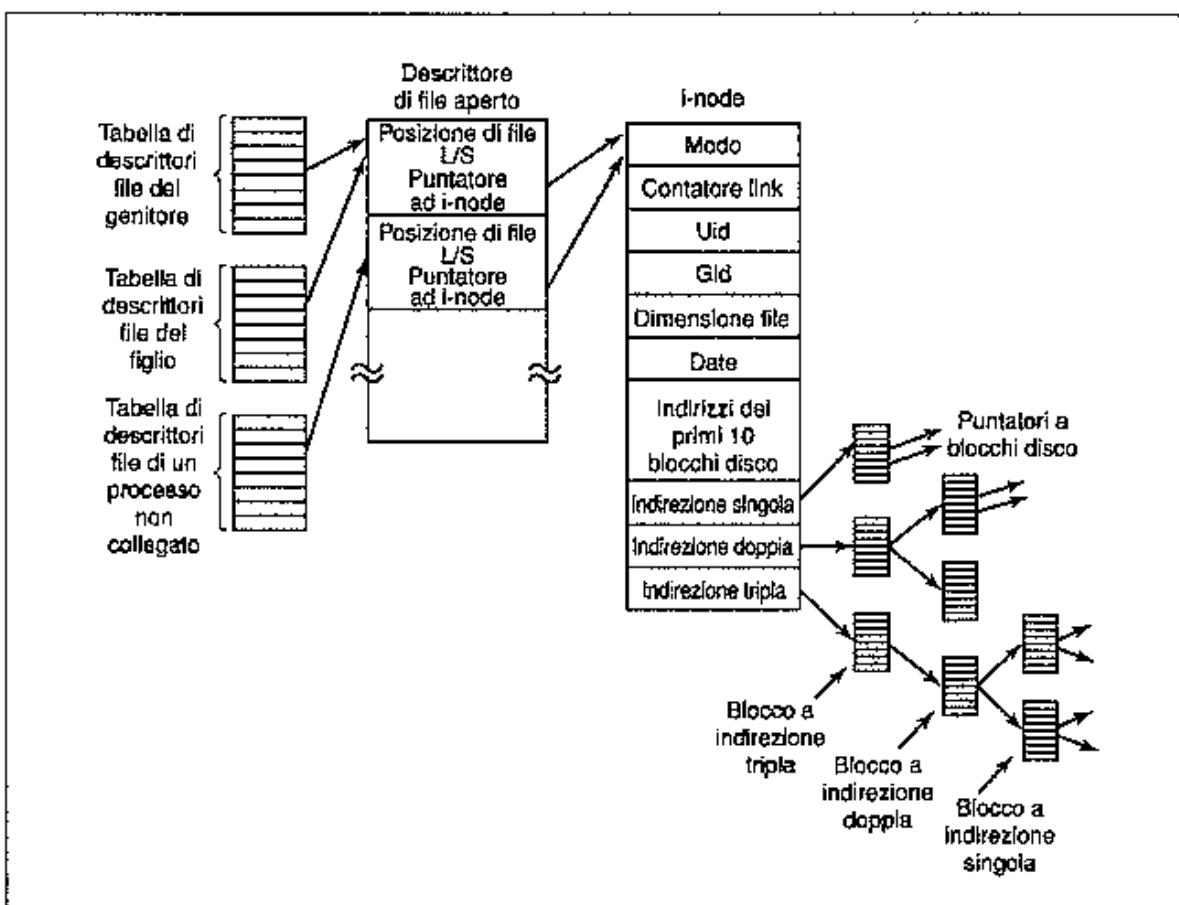


Figura 10.33 La relazione tra la tabella dei descrittori di file, la tabella di descrizione dei file aperti e la tabella degli i-node.

Ad ogni modo, se un processo non in relazione ad essi apre il file, gli viene dedicato un elemento della tabella dei descrittori dei file aperti, con la relativa posizione del file. Si deduce che il punto fondamentale della tabella dei descrittori dei file aperti consiste nel permettere ad un genitore e ad un figlio di condividere una posizione di file, e di permettere a processi non ad essi correlati di avere una loro propria posizione.

Ritornando al problema di effettuare una `read`, si è finora mostrato come sono sistematizzate la posizione del file e l'i-node. L'i-node contiene gli indirizzi del disco dei primi 10 blocchi del file; se la posizione del file rientra nei primi 10 blocchi, si legge un blocco e si copiano i dati all'utente, mentre, nel caso di grossi file, esiste un campo nell'i-node che contiene l'indirizzo di un blocco a singola indirezione, come si vede in Figura 10.33. Questo blocco contiene gli indirizzi di più blocchi; per esempio, se un blocco è di 1KB e l'indirizzo del disco è di 4 byte, il blocco a singola indirezione può contenere 256 indirizzi di disco e ciò funziona per file fino a 266KB.

Oltre tale dimensione, si usa il blocco a doppia indirezione che contiene gli indirizzi di 256 blocchi a doppia indirezione, ognuno dei quali contiene gli indirizzi di 256 blocchi di dati. Questo meccanismo è sufficiente a mantenere file fino a  $10 + 2^{16}$  blocchi (67 119 104 byte). Nel caso che anche ciò non bastasse, l'i-node ha uno spazio riservato per un blocco a tripla indirezione.

## Il Berkeley Fast File System

La descrizione precedente spiega il funzionamento classico di un file system di UNIX, mentre ora si esamineranno i miglioramenti fatti nella realizzazione di Berkeley. Innanzitutto sono state riorganizzate le directory; infatti invece di limitare i nomi dei file a 14 caratteri, esso è stato aumentato a 255. Naturalmente, l'aver cambiato la struttura di tutte le directory significa che i programmi che leggevano semplicemente le directory (che erano e sono legali) aspettandosi una sequenza di elementi di 16 byte non funzionarono più. Per permettere la portabilità sui due tipi di directory, Berkeley aggiunse le chiamate di sistema

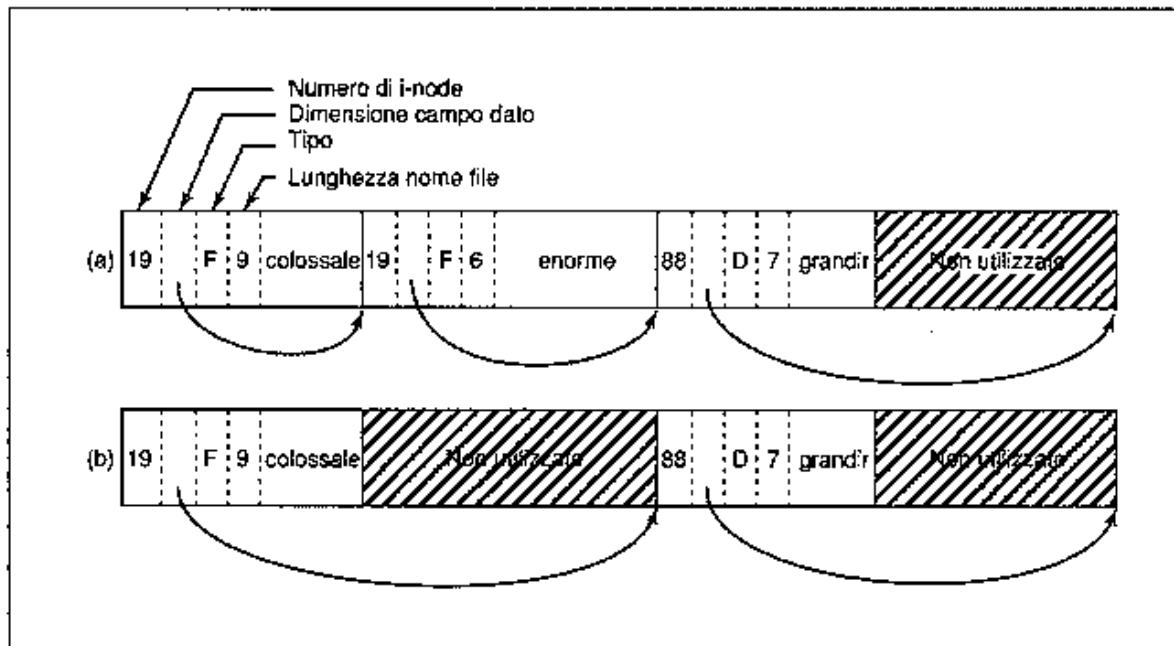


Figura 10.34 (a) Una directory BSD con tre file. (b) La stessa directory dopo la rimozione del file `enorme`.

`opendir`, `readdir`, `rewinddir` e `closedir` che permettevano ai programmi di leggere directory senza conoscerne la struttura interna. I nomi lunghi e queste chiamate di sistema, furono poi aggiunte in tutte le altre versioni di UNIX ed in POSIX.

La struttura a directory di BSD permette nomi di file lunghi fino a 255 caratteri ed è illustrata in Figura 10.34: ogni directory è formata da un certo numero di blocchi disco e le directory possono essere scritte a disco in modo atomico. All'interno delle directory, i campi dato per i file e le altre directory si trovano in modo non ordinato, uno di seguito all'altro. Tali campi dato non possono trovarsi a cavallo di blocchi disco diversi, perciò si trovano spesso byte inutilizzati alla fine del blocco.

Ogni campo dato di directory in Figura 10.34 è formato da quattro campi di lunghezza fissata ed un campo a lunghezza variabile. Il primo campo rappresenta il numero di i-node, 19 per il file *colosso*, 42 per il file *enorme* e 88 per la directory *grandit*. Di seguito si trova il campo dedicato alla dimensione della entry (in byte) compresi eventuali byte vuoti in coda. Esso serve in fase di ricerca della successiva entry, nel caso in cui al nome del file seguia un numero non conosciuto di byte vuoti, ed è rappresentato in figura dalle frecce. Successivamente, si ha il campo tipo: file, directory, eccetera L'ultimo campo a lunghezza fissa è la lunghezza del nome del file in byte (in questo esempio 9, 6 e 7). Infine, si trova il campo con il nome stesso del file, che termina con un byte 0, ed arriva fino al limite di 32-bit, sempre che non vi siano anche altri byte vuoti.

In Figura 10.34(b) si può osservare lo stato della stessa directory dopo aver rimosso il file *enorme*: tutto quello che si deve fare è aumentare la dimensione del campo dato totale di *colosso* semplicemente trasformando il precedente campo di *enorme* nella continuazione del primo.

Poiché all'interno di directory possono essere fatte solo ricerche lineari, può essere abbastanza lungo cercare un campo dato che si trova alla fine di una directory molto lunga. Per migliorare le prestazioni di ricerca, al sistema BSD è stato aggiunta la possibilità di fare la caching dei nomi: prima di guardare il nome in una directory, il sistema controlla la cache e, se presente, salta la ricerca lineare.

La seconda grande modifica fatta da Berkeley nel file system consiste nell'aver diviso i dischi in **gruppi di cilindro**, ognuno con un suo proprio superblocco, i-node e blocchi di dati; ciò fu fatto per mantenere gli i-node e i blocchi di dati di un file vicini ed evitare lunghe ricerche. Dove possibile, i blocchi sono allocati nel gruppo di cilindro che contiene l'i-node.

La terza ed ultima modifica è l'introduzione di due dimensioni per i blocchi invece che una sola; per memorizzare grossi file è più efficiente avere un piccolo numero di grossi blocchi piuttosto che molti blocchi piccoli. D'altra parte, molti file in UNIX sono piccoli, per cui avere solo blocchi grandi significherebbe una grossa perdita di spazio; se ne deduce che i due tipi di dimensionamento dei blocchi permettono un trasferimento efficiente per i file grossi ed un efficiente sfruttamento dello spazio per quelli piccoli. Il prezzo pagato per tutto ciò è una complessità maggiore nella stesura del codice.

## Il file system di Linux

Il file system iniziale di Linux era quello di MINIX, ma poiché quest'ultimo limitava il nome dei file a 14 caratteri (per essere compatibile con la Versione 7 di UNIX) e limitava la dimensione dei file a 64MB, la comunità Linux si interessò quasi immediatamente alla programmazione di un file system migliore. Il primo miglioramento si ebbe con Ext, che permetteva 255 caratteri per i nomi dei file e 2GB per dimensione, ma risultava molto più lento di MINIX, e gli sforzi continuarono. Successivamente fu inventato il file system Ext2 ottenendo, nomi lunghi per i file, file lunghi e migliori prestazioni, e così

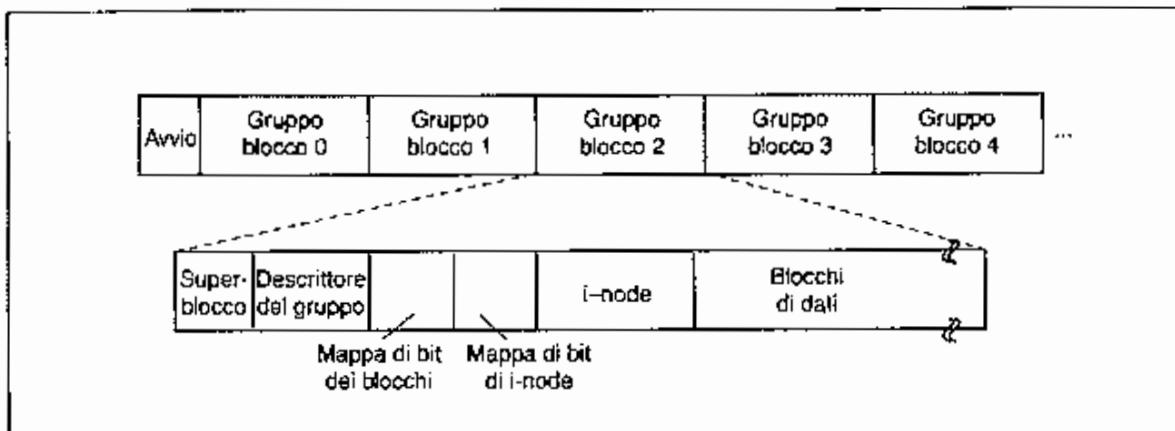


Figura 10.35 Organizzazione del file system Ext2 di Linux.

divenne presto il file system principale di Linux. Ad ogni modo, Linux supporta anche una dozzina di altri file system usando NFS (come descritto nella sezione successiva). Quando Linux viene collegato, viene proposta una scelta riguardo al file system da installare nel kernel. Altri file system, comunque possono essere aggiunti anche a tempo di esecuzione, se servono.

Ext2 risulta molto simile al file system di Berkeley con alcune piccole differenze: invece di usare i gruppi di cilindro, che sono privi di significato se si usano i moderni dischi a geometria virtuale, Ext2 (la cui organizzazione del disco è mostrata in Figura 10.35) divide il disco in gruppi di blocchi senza occuparsi se i limiti dei cilindri vengono rispettati. Ogni gruppo di blocchi incomincia con un superblocco, che dice quanti i-node e quanti blocchi sono presenti nel gruppo, la dimensione del blocco e così via. Segue il descrittore del gruppo, che rimanda alla posizione delle mappe di bit, il numero dei blocchi liberi, e degli i-node nel gruppo, ed il numero di directory. Questa informazione è molto importante visto che Ext2 tende a sparpagliare le directory per tutto il disco.

Due mappe di bit sono rispettivamente dedicate a tenere traccia dei blocchi e degli i-node liberi, ed ogni mappa è lunga un blocco. Con un blocco di 1KB, questo progetto limita un gruppo ad avere al più 8192 blocchi e 8192 i-node; il primo numero rappresenta una vera restrizione, il secondo in pratica non è un problema. Successivamente si trovano gli i-node: ogni i-node è di 128 byte, il doppio in dimensione rispetto agli i-node UNIX standard di Figura 10.32. Lo spazio extra viene utilizzato come segue: invece di permettere 10 indirizzi di blocchi diretti e 3 indiretti, Linux ne permette 12 diretti e 3 indiretti. In più, gli indirizzi sono stati estesi da 3 byte a 4 per gestire le partizioni disco più grandi di  $2^{24}$  blocchi (16GB), che era di certo un problema per UNIX. Inoltre, ci sono campi riservati per i puntatori alle liste di controllo di accessi per un maggior grado di protezione, ma questi sono ancora in fase di raffinamento. Il resto degli i-node è riservato per un uso futuro. La storia inostra che spesso bit inutilizzati non rimangono tali per tanto tempo!

Le operazioni del file system sono molto simili a quelle di Berkeley, anche se Ext2 usa blocchi da 1KB sempre, mentre Berkeley usa blocchi da 8KB che divide in blocchi da 1KB se necessario: Ext2 sceglie la strada più semplice. Come il sistema Berkeley, quando un file aumenta di dimensioni, Ext2 cerca di posizionarlo sempre nello stesso gruppo e preferibilmente con blocchi contigui ai precedenti. Anche quando viene aggiunto un nuovo file ad una directory, Ext2 cerca di posizionarlo nello stesso gruppo di blocchi della directory dove deve essere inserito. Le nuove directory vengono sparse uniformemente nel disco.

Un altro file system di Linux è il /proc (processo), nato da una idea originaria dell'ottava edizione di UNIX e più tardi copiata in 4.4BSD e System V, cui Linux ha aggiunto un sacco di idee innovative. Il concetto di base è che in /proc si crea una directory dedicata per ogni processo che viene avviato, il cui nome è il PID del processo espresso in decimale (ad esempio se un processo ha PID 619 la directory corrispondente è /proc/619). In essa vengono posti dei file che contengono informazioni sul processo, come la riga di comando, le stringhe di ambiente, le maschere dei segnali e così via. In effetti, tali file non esistono sul disco, e quando i file vengono letti, il sistema recupera le informazioni dal processo e le restituisce in un formato standard.

Molte delle estensioni di Linux sono collegate a file e directory in /proc. Esse contengono una ampia varietà di informazioni circa la CPU, le partizioni disco, i dispositivi, i vettori di interruzione, i contatori del kernel, i file system, i moduli caricati e molto altro. I programmi utente non privilegiati possono leggere molte di queste informazioni per apprendere molte informazioni sul comportamento del sistema in modo sicuro. Alcuni di questi file, inoltre, possono essere scritti in modo da variare alcuni parametri del sistema.

#### 10.6.4 NFS: il file system di rete

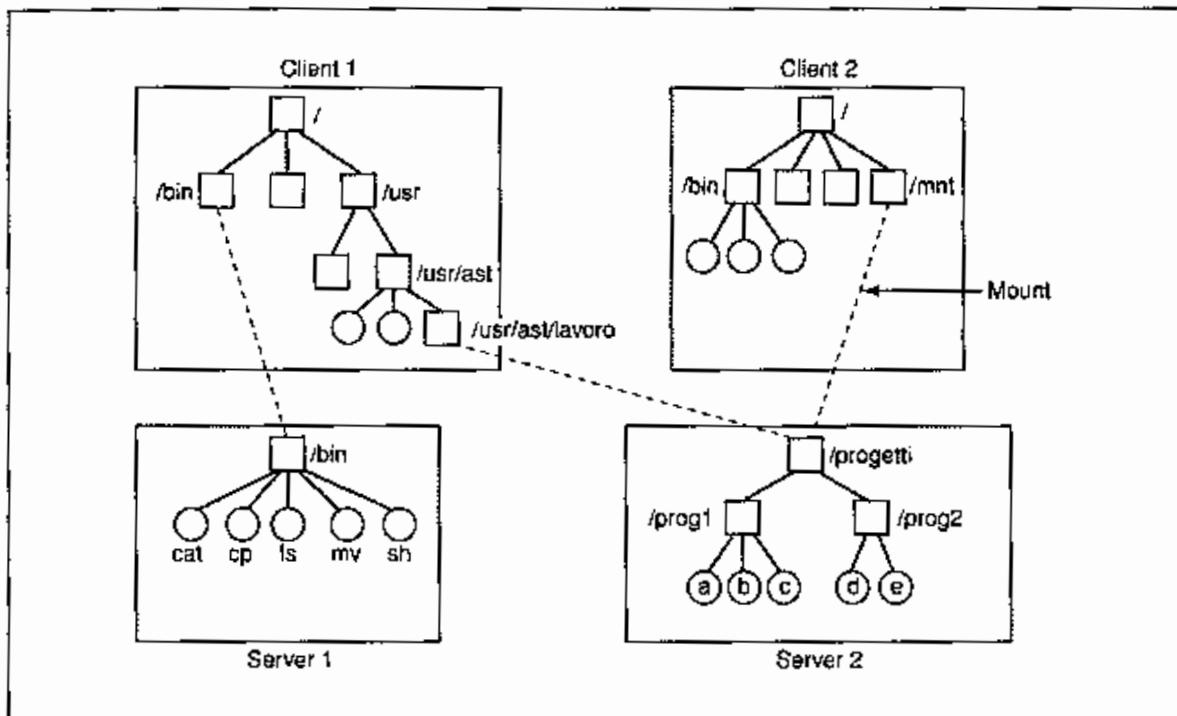
Le problematiche di rete hanno giocato da subito un ruolo importante per UNIX (la prima rete UNIX fu costruita per spostare i kernel nuovi da PDP-11/70 ad Interdata 8/32, all'epoca del porting su quest'ultimo). In questa sezione esamineremo l'**NFS** (Network file system) della Sun Microsystem, che è utilizzato da tutti i moderni sistemi UNIX (ed anche da alcuni sistemi non UNIX) per unire file system su computer separati in un unico file system logico. Esistono tre principali aspetti dell'NFS che esamineremo in dettaglio: l'architettura, il protocollo e l'implementazione.

#### L'architettura NFS

L'idea che sta alla base di NFS è quella di permettere ad una collezione arbitraria di client di condividere un file system comune. In molti casi, tutti i client ed i server si trovano sulla stessa LAN, ma non è obbligatorio. È inoltre possibile far girare l'NFS in una rete geografica, quando il server si trova molto lontano dal client. Per semplicità, vedremo il caso in cui client e server si trovano su macchine separate, anche se occorre notare che NFS permette che ogni macchina possa essere sia client sia server allo stesso tempo.

Ogni server NFS esporta una o più directory perché ad esse i client possano accedere in remoto. Quando una directory è disponibile, lo è tutto l'albero di sottodirectory sue discendenti, così di fatto si esportano unitariamente interi alberi. La lista di tutte le directory esportate viene mantenuta dal server in un file (di solito /etc/exports) in modo che le stesse directory possano essere nuovamente esportate ad ogni avvio. Dall'altro lato i client accedono alle directory inmontandole come parte del proprio file system, come mostrato in Figura 10.36.

In questo esempio il client 1 ha montato nella directory bin la directory bin del server 1, e può accedere alla shell sh digitando /bin/sh, e ricuperandola dal server 1. Le workstation prive di disco spesso hanno in RAM uno scheletro di file system, e prendono tutti i file dai server remoti in questo modo. Similmente lo stesso client ha montato nella directory /usr/ast/lavoro la directory /progetti del server 2 e quindi può accedere al file a con il cammino /usr/ast/lavoro/prog1/a. Infine, il client 2 ha montato in /mnt la directory /progetti sempre dal server 2 e può accedere al file a con il nome /mnt/prog1/a. Come si vede, lo stesso file può avere nome diverso se la directory a cui appartiene viene montata in un



**Figura 10.36** Esempi di file system remoti montati; le directory sono mostrate come quadrati ed i file come cerchi.

posto diverso nei rispettivi alberi di due client. Il punto di montaggio è completamente locale al client: il server non conosce nulla di questo.

## I protocolli NFS

Visto che uno degli scopi principali di NFS è quello di supportare sistemi eterogenei con client e server che possiedono anche file system differenti, su hardware differente, è essenziale che in esso siano ben definite le interfacce client-server: solo così ci si può aspettare che l'implementazione di un nuovo client possa facilmente funzionare con i vecchi server e viceversa.

NFS risponde a tali richieste definendo due protocolli client-server. In generale, un **protocollo** è un insieme di richieste mandate da un client ai server, più le corrispondenti risposte dei server alle interrogazioni.

Il primo protocollo NFS gestisce il mounting di directory: un client spedisce ad un server un pathname ed una richiesta di permessi per montare tale directory. Per il server, il posto in cui la directory verrà montata non è importante, quindi non fa parte della richiesta. Se il pathname è valido e la directory richiesta è stata esportata, il server restituisce un **file handle** (gestore di file) da assegnare al client. Il gestore contiene dei campi che determinano univocamente il tipo di file system, il disco, il numero di i-node della directory e le informazioni di sicurezza; esso verrà inoltre utilizzato per ogni successiva operazione di lettura o scrittura nella directory montata o nelle sue sottodirectory.

Quando UNIX viene avviato, esso esegue tra le altre cose lo script di shell */etc/rc* prima di andare in modalità multiutente. I comandi per montare i file system remoti possono essere messi all'interno di questo script in modo da montare tutto il necessario prima di qualsiasi accesso. In modo alternativo alcune versioni UNIX supportano l'**automount**.

ting che permette di associare un insieme di directory remote ad una directory locale senza che queste siano montate in fase di avvio, e neppure i loro server vengono contattati. Ma, la prima volta che un file remoto viene aperto, il sistema operativo manda un messaggio ad ognuno dei server ed il primo che risponde vince e la directory relativa viene montata.

L'automounting ha due vantaggi fondamentali rispetto al montaggio statico tramite il file */etc/rc*: in primo luogo se uno dei server NFS chiamato da */etc/rc* fosse giù, è impossibile portare a termine l'avvio del client, opporre lo si fa ma con non poche difficoltà, ritardi e un po' di messaggi di errore; se l'utente in quel momento non necessita del server fuori uso, tutto il lavoro risulta sprecato. In secondo luogo, permettendo ai client di provare un insieme di server in parallelo, si può ottenere un certo grado di tolleranza ai guasti e le prestazioni possono essere migliorate (scegliendo il primo server che risponde, è presumibilmente quello meno carico).

D'altro canto è tacitamente ipotizzato che tutti i file system alternativi dell'automounting sono identici. Poiché NFS non fornisce alcun supporto per la replicazione di file o directory, è compito dell'utente organizzare tutti i file system in modo da essere gli stessi. Conseguentemente, l'automounting è spesso utilizzato solo per file system in sola lettura contenenti binari di sistema ed altri file che cambiano raramente.

Il secondo protocollo NFS è quello dedicato agli accessi a file e directory. Un client manda messaggi ai server per manipolare directory e leggere e scrivere file; inoltre, può accedere ad attributi di file, come il modo, la dimensione ed la data dell'ultima modifica. Quasi tutte le chiamate di sistema di UNIX sono supportate da NFS, tranne sorprendentemente *open* e *close*.

La mancanza del supporto a queste due chiamate non è un caso: non è necessario aprire un file prima di leggerlo e neanche occorre chiuderlo una volta concluso il lavoro. Di contro, per leggere un file, un client manda al server un messaggio di *lookup* contenente il nome del file, con la richiesta di cercarlo e restituire il gestore di file, la struttura che identifica il file (ovvero la struttura che contiene l'identificatore di file system, ed il numero di i-node, tra gli altri dati). Diversamente da quello che avviene per una chiamata *open*, questa operazione di *lookup* non copia alcuna informazione dentro le tabelle interne di sistema. La chiamata *read* contiene il gestore di file e l'offset del file da leggere, ed il numero di byte di cui si ha bisogno. Ogni messaggio spedito dal client è auto-contenuto, ed il vantaggio di questo schema è che il server non ha bisogno di ricordarsi nulla sulle connessioni aperte fra le chiamate. Quindi, se un server va in crash e in seguito viene sistemato, nessuna informazione sui file aperti va persa, poiché di fatto non esistono informazioni da poter perdere; un server di questo tipo, che non mantiene informazioni di stato sui file aperti è chiamato stateless (senza stati).

Sfortunatamente, il metodo di NFS rende difficoltosa la consistenza con la semantica UNIX. Per esempio, in UNIX un file può essere aperto e bloccato (*locked*) in modo che altri processi non possano accedervi; quando il file viene chiuso il lock viene liberato. In un server stateless come NFS, invece, i lock non possono essere associati a file aperti, poiché il server non conosce quali essi siano. NFS quindi necessita di un meccanismo separato supplementare per gestire le operazioni di locking.

Inoltre, NFS si serve del meccanismo standard UNIX di protezione dei file, con i bit *rwx* per il proprietario, il gruppo e gli altri, rispettivamente (di cui si è parlato nel capitolo 1 e che verranno descritti in dettaglio di seguito). Originariamente, ogni messaggio di richiesta conteneva semplicemente gli ID dell'utente e del gruppo del chiamante che venivano poi adoperati dall'NFS per validare l'accesso. In effetti, NFS aveva un po' troppa fiducia che il client fosse onesto. Dopo molti anni di esperienza si è capito che tale assunzione era - come dire? - da semplicotti. Allo stato attuale delle cose, può essere utilizzata la crittografia a chiave pubblica per stabilire una chiave sicura per la validazione

del client e del server per ogni richiesta ed ogni risposta. Quando tale opzione è abilitata, un client disonesto non è in grado di impersonare un altro client onesto poiché non conosce la chiave privata di quest'ultimo.

## Implementazione dell'NFS

Sebbene l'implementazione del codice client-server sia indipendente dal protocollo NFS, la maggior parte dei sistemi UNIX utilizza una implementazione a tre livelli simile a quella mostrata in Figura 10.37. Lo strato superiore è lo strato delle chiamate di sistema, ed è responsabile di chiamate come `open`, `read` e `close`. Dopo aver analizzato la chiamata e fatto il controllo sui parametri, viene chiamato il secondo strato - lo strato del File System Virtuale (VFS).

Il suo compito è di mantenere una tabella con un campo dato per ogni file aperto, analogamente alla tabella degli i-node per i file aperti in UNIX. Nello standard UNIX, un i-node viene determinato univocamente dalla coppia (dispositivo, numero di i-node); nel VFS, invece, ogni campo dato è chiamato v-node (i-node virtuale), ed è associato ad ogni file aperto. I v-node mantengono le informazioni sui file locali e remoti: nel caso di file remoti, vi sono abbastanza informazioni per essere in grado di accedervi mentre, per i file locali, vengono registrati il file system e l'i-node visto che nei sistemi UNIX moderni possono coesistere più file system (ad esempio V7, Berkeley FFS, Ext2FS, /proc, FAT, eccetera). Sebbene VFS sia stato inventato per supportare NFS, tutti i sistemi moderni lo utilizzano anche se non fanno uso di NFS.

Per vedere in dettaglio come vengono utilizzati i v-node, proviamo a tracciare una sequenza di chiamate di sistema `mount`, `open` e `read`. Per montare un file system remoto:

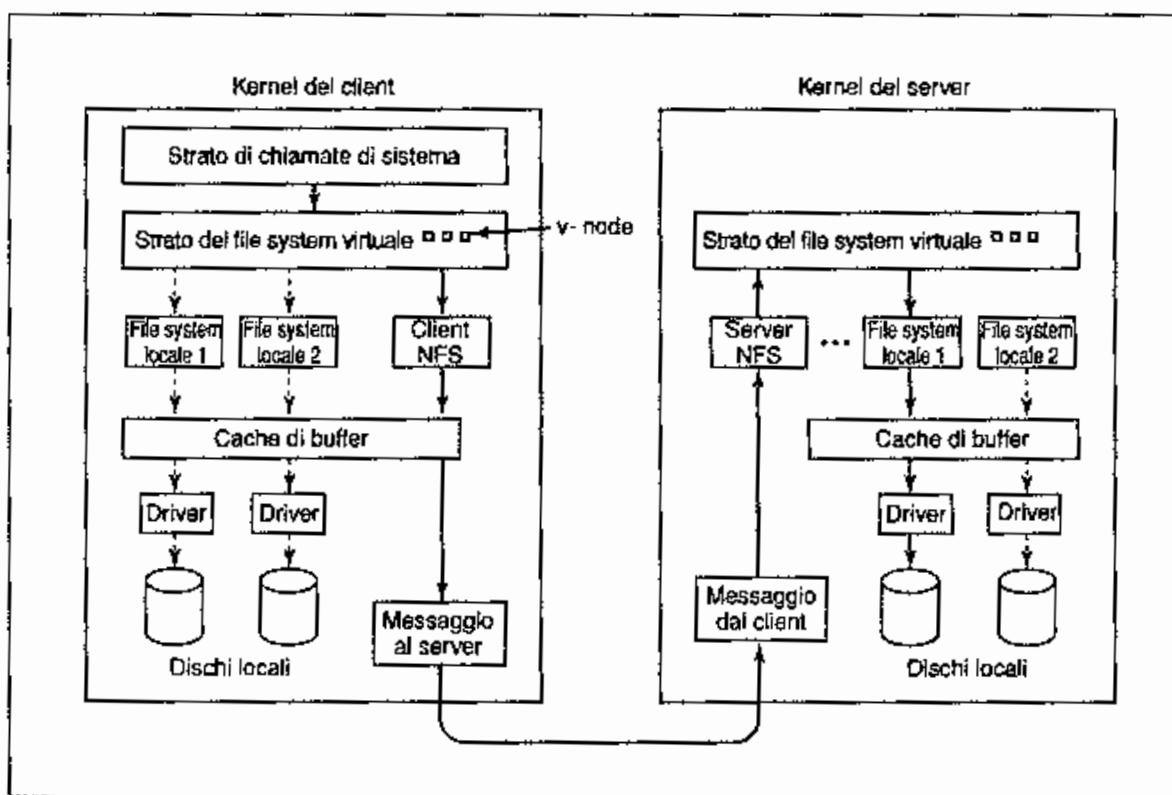


Figura 10.37 La struttura a strati dell'NFS.

to, l'amministratore di sistema (o `/etc/rc`) chiama il programma `mount` specificando la directory remota, la directory locale sulla quale deve essere montata ed altre informazioni. Il programma `mount` analizza il nome della directory remota da montare, e scopre il nome del server NFS a cui appartiene. Quindi contatta la macchina per chiedergli il gestore di file relativo; se la directory esiste ed è disponibile per il mounting remoto, il server restituisce il gestore di file per tale directory. Infine, il programma fa la chiamata di sistema `mount` passando il gestore di file al kernel.

Successivamente il kernel costruisce un v-node per la directory remota e chiede al codice del client NFS (in Figura 10.37) di creare un r-node (i-node remoto) nelle sue tabelle interne in modo da tenervi il gestore di file associato alla directory, ed il v-node punta all'r-node. In effetti, ogni v-node nello strato VFS conterrà o un puntatore all'r-node nel codice client NFS o un puntatore ad un i-node in uno dei file system locali (mostrato con linee tratteggiate in Figura 10.37). Per questo dai v-node è possibile osservare immediatamente se un file o una directory è locale o remoto: se è locale è possibile recuperare subito il corretto file system ed il relativo i-node, se remoto invece, il v-node contiene le informazioni per recuperare l'host remoto ed il gestore di file.

Quando un file remoto viene aperto da un client, ad un certo punto del pathname fornito, il kernel riconosce la directory su cui è montata la directory remota, ed utilizza il v-node per recuperare il puntatore all'r-node associato. Quindi chiede al client NFS di aprire il file: questi legge la rimanente porzione del pathname sul server remoto associato alla directory montata e restituisce un gestore di file. Inoltre costruisce un r-node per il file remoto e lo restituisce allo strato VFS, che lo pone nella propria tabella in un v-node (che punta a quell'r-node). Di nuovo, si può osservare che ogni file aperto o directory ha un v-node che punta ad un r-node o ad un i-node.

Successivamente, al richiedente viene fornito un descrittore di file per il file remoto, che è mappato sul v-node da tabelle nello strato VFS. Si noti che non vengono costruite tabelle lato server: sebbene il server sia in grado di fornire gestori di file su richiesta, esso non tiene traccia di quali file abbiano gestori e quali no. Quando un gestore di file viene spedito al server per un accesso ad un certo file, il server controlla il gestore e, se valido, lo usa. Il processo di validazione può comprendere un'autenticazione di chiave contenuta nelle intestazioni RPC, se la sicurezza di sistema è attiva.

Quando un descrittore di file viene utilizzato in una successiva chiamata di sistema, per esempio una `read`, lo strato VFS trova il corrispondente v-node e da questo determina se si tratta di un file locale o remoto e quale sia l'i-node (o r-node) associato; quindi spedisce un messaggio al server contenente il gestore, l'offset del file (mantenuto a lato client e non server), ed il numero dei byte. Per ragioni di efficienza del sistema, i trasferimenti tra client e server sono fatti mediante grandi pezzi di informazioni, di solito 8192 byte, anche se in realtà sono richiesti solo pochi byte.

Quando al server arriva una richiesta, questa viene direttamente passata allo strato VFS che determina quale file system locale detiene il file cercato. Quindi effettua una chiamata al file system locale per leggere e restituire i byte, ed infine questi dati vengono mandati al client. Dopo che lo strato VFS del client ha ricevuto un pezzo di 8KB di dati, che aveva chiesto, esso automaticamente invia una nuova richiesta dei blocchi di informazione successivi, così gli arriveranno prima, nel caso in cui fossero richiesti anche essi. Questa politica, detta lettura in avanti o anche `read ahead`, migliora notevolmente le prestazioni del sistema.

Nel caso di operazioni di scrittura il procedimento è analogo: i trasferimenti vengono sempre fatti a pezzi di 8KB. Se una chiamata di sistema `write` fornisce meno di 8KB di dati, i dati vengono accumulati localmente: l'intero pezzo di 8KB viene mandato effettivamente al server solo quando è completamente pieno. Ad ogni modo, quando il file viene chiuso, tutti i dati accumulati vengono comunque spediti subito al server.

Un'altra tecnica utilizzata per migliorare le prestazioni del sistema è la cache, come nel caso dello standard UNIX: i server pongono in una cache i dati per evitare gli accessi a disco. Tutto ciò è invisibile agli occhi del client che, invece, mantengono due cache, una per gli attributi dei file (i-node) e l'altra per i dati dei file. Quando un i-node o un certo blocco di dati è necessario, si controlla se la richiesta può essere soddisfatta dalla cache; se sì, il traffico di rete può essere limitato.

Purtroppo però la caching dei client porta oltre al guadagno di prestazioni, anche un certo numero di problemi fastidiosi. Si supponga ad esempio che due client stiano tenendo in cache lo stesso blocco di file, e che uno dei due lo stia modificando. Quando l'altro client legge il blocco ottiene dalla cache il vecchio valore: la cache non è coerente.

Data la potenziale gravità del problema, l'implementazione di NFS cerca di operare in vari modi per mitigarlo: innanzitutto, vi è un timer associato ad ogni cache, e quando il timer scade, il campo dato viene buttato via; normalmente il timer è di circa 3 secondi per blocchi di dati e 30 secondi per blocchi di directory, e tale uso dei timer riduce il rischio di inconsistenza. In più, quando un file che si trova nella cache viene aperto, viene spedito un messaggio al server per scoprire quando è l'ultima volta che tale file è stato modificato. Se l'ultima modifica è avvenuta quando il file era già nella cache, la copia del file in cache viene buttata, e si richiede al server il nuovo file, in modo da mantenere la consistenza dei dati. Infine, ogni 30 secondi il timer di cache scade e tutti i blocchi sporchi (cioè modificati) vengono rispediti al server. Sebbene non perfette, queste toppe rendono il sistema utilizzabile nella maggior parte delle circostanze.

## 10.7 La sicurezza in UNIX

A parte il nome, UNIX è sempre stato un sistema multiutente, quindi la sicurezza ed il controllo delle informazioni sono stati sempre argomenti di fondamentale importanza. Nelle seguenti sezioni vedremo alcuni degli aspetti di sicurezza di UNIX.

### 10.7.1 Concetti di base

La comunità di utenti di un sistema UNIX è formata da un certo numero di utenti registrati, ognuno con un unico UID (User ID). Un UID è un intero tra 0 e 65535 ed i file (ma anche i processi ed altre risorse) vengono marcati con l'UID dal relativo proprietario. Per difetto, il proprietario di un certo file è la persona che lo ha creato, sebbene vi siano modi di cambiare tale proprietà.

In UNIX inoltre, gli utenti possono essere organizzati in gruppi numerati con interi a 16 bit chiamati GID (Group ID). L'assegnazione di un utente ad un certo gruppo viene fatto manualmente dall'amministratore di sistema, che crea dei campi dato nel database del sistema che mettono in relazione un certo gruppo con un certo utente. Originariamente, un utente poteva appartenere ad un solo gruppo, ma alcune ultime versioni di UNIX permettono di far parte di più gruppi nello stesso momento. Per semplicità non parleremo più di questo caso qui di seguito.

Il meccanismo di sicurezza alla base di UNIX è abbastanza semplice. Ogni processo porta con sé l'UID ed il GID del proprietario: quando un file viene creato eredita l'UID ed il GID dal processo che lo ha generato. Il file recupera anche un certo insieme di permessi determinati dal processo creatore. Questi permessi specificano che tipo di accessi possono essere fatti dal proprietario del file, dal gruppo a cui appartiene e da tutti gli altri, per ognuna di questa categorie è possibile specificare permessi in lettura, scrittura

| Binario    | Simbolico | Accessi permessi al file                                                                                      |
|------------|-----------|---------------------------------------------------------------------------------------------------------------|
| 1110000000 | rwx-----  | Il proprietario può leggere, scrivere ed eseguire il file                                                     |
| 1111110000 | rwxrwx--- | Il proprietario ed il gruppo possono leggere, scrivere ed eseguire il file                                    |
| 1101000000 | rw-r----  | Il proprietario può leggere, scrivere; il gruppo può leggere                                                  |
| 1101001000 | rw-r--r-- | Il proprietario può leggere, scrivere; tutti possono leggere                                                  |
| 111101101  | rwxr-xr-x | Il proprietario può leggere, scrivere ed eseguire il file; gruppo e altri possono leggere ed eseguire il file |
| 0000000000 | -----     | Nessuno ha diritti sul file                                                                                   |
| 000000111  | ----rwx   | Solo gli altri hanno accesso (strano ma possibile)                                                            |

Figura 10.38 Alcuni esempi di modi di protezione di file.

ed esecuzione, tramite le lettere *r*, *w* e *x*, rispettivamente. La possibilità di eseguire un certo file, ha senso, ovviamente, se il file è un binario eseguibile e quindi, un tentativo di eseguire un file non eseguibile (cioè che non inizia con una intestazione valida) restituirà un errore. Con tre permessi e tre categorie, bastano 9 bit per rappresentare i possibili diritti; alcuni esempi di diritti sono mostrati in Figura 10.38.

Le prime due righe sono banali: forniscono completo accesso al proprietario ed al gruppo, rispettivamente. La terza riga dà il permesso di lettura del file al gruppo e lettura/scrittura al proprietario; la quarta è tipica di un file che l'utente vuole rendere pubblico. Similmente la quinta riga concede i permessi standard per un programma eseguibile. La sesta riga, infine, non concede nessun permesso a nessun utente: questo può servire per file dummy, usati per implementare la mutua esclusione. Poiché un tentativo di creare un tale file fallirà se esso già esiste, se più processi cercassero di crearlo contemporaneamente, ci riuscirà uno solo. L'ultimo esempio è veramente bizzarro, perché dà più permessi al resto del mondo che al proprietario, ma le regole di protezione lo ammettono. Fortunatamente, un proprietario può in seguito cambiare il modo di protezione, anche se non vi ha più accesso.

L'utente con UID pari a 0 è un utente speciale, detto **superuser** (o **root**, radice). Il superuser ha il potere di leggere e scrivere su tutti i file del sistema, a prescindere dal proprietario e dal modo di protezione. I processi con UID 0 hanno la possibilità, inoltre, di far alcune chiamate di sistema protette, che i processi ordinari non possono fare. Normalmente, solo l'amministratore di sistema conosce la password di root, sebbene molti studenti considerino un grande divertimento cercare di entrare nel sistema tramite qualche baco di sicurezza, senza conoscere la password di superuser. I gestori solitamente non approvano queste attività.

Le directory sono file ed hanno lo stesso modo di protezione, eccetto il fatto che la *x* si riferisce a permessi di ricerca nella directory piuttosto che all'esecuzione. Quindi una directory con modo *rwxr-rr-x* permette al proprietario di leggerla, scriverla e cercarvi dentro ed al gruppo ed al resto degli utenti la lettura e la ricerca, ma non aggiungere né rimuovere file.

I file speciali corrispondenti a dispositivi di I/O hanno gli stessi bit di protezione: questo meccanismo può essere utilizzato per limitare gli accessi ai dispositivi di I/O. Ad esempio, il file speciale della stampante, */dev/lp*, può bloccare il permesso di utilizzo della stampante a tutti tranne che al proprietario (ad esempio il superuser, oppure il demone di stampa, di proprietà di un utente speciale) con una sequenza di bit come *rw-----*. Dopo tutto, se tutti potessero stampare il caos sarebbe quasi certo.

Di certo, se il proprietario tosse il demone di stampa con la protezione come sopra,

solo esso potrebbe stampare, e ciò sicuramente salverebbe molti giovani alberi da una morte prematura. Ma potrebbe succedere che un utente abbia reale necessità di stampare un documento, più in generale c'è un problema di permettere accessi controllati a tutti i dispositivi di I/O ed alle risorse del sistema.

Una possibile soluzione è l'inserimento, oltre ai 9 bit, di un nuovo bit di protezione, **SETUID**: quando un programma con tale bit attivato viene eseguito, l'UID effettivo per quel processo diventa l'UID del proprietario del file eseguibile, piuttosto dell'UID dell'utente che lo ha chiamato. Quando un processo cerca di aprire un file, viene controllato il suo UID effettivo e non l'UID sottostante, quindi, se il programma che accede alla stampante è di proprietà del demone, ma ha il bit SETUID impostato, tutti gli utenti possono eseguirlo, con i privilegi del demone (quindi possono aprire */dev/lp*) ma solo quando eseguono tale programma (che probabilmente accorderà i job di stampa e li farà uscire in modo ordinato).

Molti programmi UNIX sono posseduti dal superuser ma con SETUID attivo. Per esempio, il programma che permette agli utenti di cambiare la propria password, *passwd*, necessita di scrivere nel file di password: rendere il file di password pubblicamente scrivibile sarebbe un errore madornale. Invece, *passwd* è un programma di proprietà di *root* con SETUID attivo: sebbene abbia completo accesso al file di password, cambierà solamente la password dell'utente chiamante e non permetterà altri accessi al file.

In aggiunta al bit SETUID vi è anche il bit SETGID che funziona analogamente ma per i gruppi, dando all'utente momentaneamente il GID effettivo del programma; in pratica, però questo bit viene usato molto raramente.

## 10.7.2 Chiamate di sistema di sicurezza in UNIX

Esistono poche chiamate di sistema in UNIX dedicate alla sicurezza, le più importanti sono quelle elencate in Figura 10.39. La più usata è senza dubbio *chmod*, per cambiare i bit di protezione di un file o di una directory. Per esempio

```
s = chmod("/usr/ast/nuovogioco", 0755);
```

definisce i bit di protezione del file *nuovogioco* come *rwxr-xr-x* (si noti che 0755 è una costante ottale, utile poiché i bit sono a gruppi di tre). Solo il proprietario del file ed il superuser possono cambiare i bit di protezione.

La chiamata *access* fa un controllo per vedere se l'accesso tentato è effettivamente per-

| Chiamata di sistema                                | Descrizione                                         |
|----------------------------------------------------|-----------------------------------------------------|
| <code>s = chmod(path, modo)</code>                 | Cambia i parametri di protezione di un file         |
| <code>s = access(path, modo)</code>                | Controlla gli accessi considerando i veri UID e GID |
| <code>uid = getuid()</code>                        | Recupera il vero UID                                |
| <code>uid = geteuid()</code>                       | Recupera l'UID effettivo                            |
| <code>gid = getgid()</code>                        | Recupera il vero GID                                |
| <code>gid = getegid()</code>                       | Recupera il GID effettivo                           |
| <code>s = chown(path, proprietario, gruppo)</code> | Cambia il proprietario ed il gruppo                 |
| <code>s = setuid(uid)</code>                       | Definisce l'UID                                     |
| <code>s = setgid(gid)</code>                       | Definisce il GID                                    |

Figura 10.39 Alcune chiamate di sistema legate alla sicurezza. Il valore restituito per *s* è -1 in caso di errore; *uid* e *gid* sono rispettivamente UID e GID. Gli altri parametri sono autoesplicativi.

messo in base al UID ed al GID reali. Questa chiamata è necessaria per evitare buchi di sicurezza in programmi che hanno SETUID attivo e sono, ad esempio, posseduti dal superuser: di fatto questi programmi possono fare ogni cosa. Talvolta però è necessario chiedersi se l'utente può attuare un certo accesso. Il programma non può semplicemente provare perché, nelle condizioni in cui è, ogni accesso avrebbe successo. Con la chiamata `access` il programma può scoprire se l'accesso è permesso usando UID e GID reali dell'utente.

Le successive quattro chiamate di sistema restituiscono gli UID e GID reali ed effettivi, e le ultime tre chiamate sono messe solo a *root*: esse cambiano il proprietario del file e l'UID/GID di un processo.

### 10.7.3 Implementazione della sicurezza in UNIX

Quando un utente accede al sistema, il programma di login, `login` (che ha SETUID ed è di *root*) chiede un nominativo di accesso ed una password, quindi fa un hash della password e la ricerca nel file di password `/etc/passwd` per vedere se esiste (i sistemi in rete lavorano in modo leggermente diverso). La ragione per la quale viene utilizzato un hash è quella di impedire di memorizzare la password in forma non crittata in un qualche posto del sistema. Se la password è corretta, il programma di login guarda in `/etc/passwd` per vedere quale sia la shell preferita dell'utente, ad esempio `sh`, o magari un'altra shell come `csh` o `ksh`. Quindi utilizza le chiamate di sistema `setuid` e `setgid` per assegnarsi UID e GID dell'utente (infatti in partenza era SETUID del superuser). Dopo, apre la tastiera per standard input (descrittore di file 0), lo schermo come standard output (descrittore di file 1) e sempre lo schermo come standard error (descrittore di file 2). Infine esegue la shell preferita e termina.

A questo punto la shell è in esecuzione con il corretto UID e GID e con gli standard I/O inizializzati correttamente; tutti i processi che essa produce mediante `fork` (cioè i comandi dati dall'utente) ereditano in modo automatico l'UID ed il GID, e possiederanno il corretto proprietario e gruppo. Gli stessi valori sono ereditati da ogni file che creano.

Quando un qualsiasi processo prova ad aprire un file, il sistema prima controlla i bit di protezione del file richiesto (che sono nell'i-node) per vedere se l'accesso è permesso alla UID e GID effettiva del chiamante; se sì, il file viene aperto e si restituisce il descrittore di file; se no, il file non viene aperto e viene restituito -1. Non vengono fatti altri controlli su successive chiamate di `read` e `write`: come conseguenza, se i bit di protezione cambiano mentre un file è aperto, il nuovo modo avrà effetto solo dopo la chiusura del file.

Per concludere, la sicurezza in Linux è da considerarsi essenzialmente la stessa di quella di UNIX: tutte le caratteristiche sono implementate, e c'è poco di altro da fare.

## 10.8 Sommario

UNIX nacque come un sistema timesharing di minicomputer, ma attualmente è usato su una gamma di computer che va dal notebook ai supercomputer. Esso è composto da tre interfacce: la shell, la libreria di chiamate di sistema C e le chiamate di sistema vere e proprie. La shell permette agli utenti di digitare comandi per l'esecuzione; questi possono essere comandi semplici, pipeline o strutture più complesse. L'input e l'output possono essere ridiretti. La libreria C contiene le chiamate di sistema e altre chiamate di più alto livello, come `printf` per scrivere output formattato nei file. L'interfaccia di chiamate di sistema effettiva conta circa 100 chiamate ognuna delle quali fa ciò che deve e nulla di più.

I concetti chiave in UNIX comprendono il processo, il modello della memoria, il file system e l'I/O. I processi possono creare dei sottoprocessi, dando origine ad un albero di processi, la gestione dei processi in UNIX si basa su due strutture dati, la tavola dei processi e la struttura utente, la prima sempre in memoria, la seconda invece può essere messa su disco o paginata. La creazione dei processi si fa duplicando la tavola del processo creatore, ed in seguito la sua immagine di memoria; la schedulazione usa un algoritmo basato su priorità che favorisce i processi interattivi.

Il modello di memoria è formato da un segmento di testo, uno dei dati ed uno dello stack, per ciascuno dei processi. La gestione della memoria si faceva inizialmente mediante lo swapping, ma ora, nella maggior parte dei sistemi UNIX si usa la paginazione. La core map tiene traccia dello stato di ogni pagina ed il demone di pagina usa l'algoritmo dell'orologio per mettere a disposizione un numero sufficiente di pagine libere.

I dispositivi di I/O sono strutturati mediante file speciali ciascuno con un numero di major e minor device. I dispositivi di I/O a blocchi usano un buffer cache per ridurre il numero di accessi al disco; per gestire la cache viene usato un algoritmo LRU. L'I/O di caratteri può essere effettuato usando la modalità raw o cooked, caratteristiche utili per l'I/O di caratteri sono la disciplina di riga e gli stream (flussi).

Il file system è gerarchico con file e directory; tutti i dischi sono montati in un singolo albero di directory che parte da una unica radice. I file individuali possono essere collegati in directory ovunque nel file system. Per utilizzare un file, deve essere prima aperto, ottenendo un descrittore di file da utilizzare nelle letture e scritture successive. Internamente il file system usa tre tabelle principali: la tabella dei descrittori dei file, la tabella dei descrittori dei file aperti e la tabella degli i-node; quest'ultima è la più importante di tutte e contiene le informazioni amministrative del file e la locazione dei suoi blocchi.

La protezione è basata sui bit di controllo di lettura, scrittura ed esecuzione da parte del proprietario, del gruppo e di tutti gli altri. Per le directory il bit per l'esecuzione è da interpretarsi in termini di diritto alla ricerca.

## PROBLEMI

- Quando il kernel prende una chiamata di sistema, come capisce quale deve essere la chiamata di sistema da recuperare?
- Una directory contiene i seguenti file:

|          |           |          |          |         |
|----------|-----------|----------|----------|---------|
| aardvark | ferret    | koala    | porpoise | unicorn |
| bonefish | gruunion  | llama    | quacker  | vicuna  |
| capybara | hyena     | marmot   | rabbit   | weasel  |
| dingo    | ibex      | nuthatch | seahorse | yak     |
| emu      | jellyfish | ostrich  | tuna     | zebu    |

Quali file verranno elencati dopo il comando:

`ls [abc]*e*?`

- Quale effetto produce la seguente pipeline in UNIX?  
`grep nd xyz | wc -l`
- Scrivere una pipeline UNIX che stampi l'ottava riga del file z sullo standard output.

5. Perché UNIX fa distinzione tra standard output e standard error quando entrambi hanno come default il terminale?
6. Si supponga che un utente digitò a terminale i seguenti comandi:  

```
a|b|c&
d|e|f&
```

Dopo che la shell li ha elaborati, quanti sono processi in esecuzione?
7. Quando fa partire un comando, la shell di UNIX definisce le copie delle sue variabili di ambiente, come *HOME*, sullo stack del processo così il processo può sapere qual è la sua directory di provenienza. Se questo processo in un secondo momento facesse una *fork*, il figlio potrà avere a disposizione queste variabili automaticamente?
8. Quanto tempo è necessario ad un processo per fare una *fork* di un processo figlio nelle seguenti ipotesi: dimensione del testo = 100KB, dimensione dei dati = 20KB, dimensione dello stack = 10KB, dimensione della tabella di processo = 1KB; struttura utente = 5KB. L'avvio e il ritorno dal kernel richiede 1 ms, la macchina può copiare ogni 50 ns una parola di 32 bit, ed i segmenti di testo sono condivisi.
9. Appena divennero comuni programmi con dimensioni dell'ordine dei megabyte, il tempo necessario per una chiamata di sistema *fork* crebbe in proporzione e, ciò che è peggio, questo tempo era davvero sprecato, perché tanti programmi, dopo la *fork* chiamavano la *exec*. Per migliorare le prestazioni, a Berkeley fu inventata una nuova chiamata di sistema *vfork*, con l'effetto che il figlio condivideva con il genitore lo spazio degli indirizzi, invece di farne una copia. Descrivere una situazione in cui un figlio dal comportamento poco furbo fa qualcosa che rende la semantica della *vfork* fondamentalmente diversa da quella della *fork*.
10. Se un processo UNIX possiede un valore di *uso\_della\_CPU* di 20, dopo quanti intervalli di tempo  $\Delta T$  tale valore decade a zero, se non viene schedulato per un certo tempo? Si utilizzi il semplice algoritmo di decadimento dato nel testo.
11. Ha senso recuperare la memoria di un processo quando entra in uno stato zombie? Perché o perché no?
12. A quale concetto hardware è strettamente collegato un segnale? Si diano due esempi di segnali.
13. Perché pensi che i progettisti di UNIX abbiano reso impossibile a un processo mandare un segnale ad un altro processo non appartenente al suo gruppo?
14. Una chiamata di sistema viene di solito implementata utilizzando una istruzione di interrupt di software (trap). Si potrebbe utilizzare una chiamata di procedura ordinaria sull'hardware Pentium? Se sì, in quali condizioni e come? Se no, perché?
15. In generale, pensi che i demoni abbiano priorità maggiore o minore dei processi interattivi? Perché?
16. Quando un processo opera una *fork*, al figlio deve essere assegnato come PID un intero unico nel sistema. È sufficiente avere un contatore nel kernel che viene incrementato per ogni processo creato, ed usare tale valore del contatore come PID? Si discuta la risposta.
17. In ogni campo dato processo nella tabella dei processi, viene immagazzinato il PID del processo padre. Per quale motivo?

18. Quale combinazione della mappa di bit `sharing_flags` di un comando `clone` di Linux corrisponde alla chiamata convenzionale `fork` di UNIX? E per creare un thread UNIX convenzionale?
19. Lo schedulatore Linux computa la bontà (goodness) per i processi in tempo reale sommando 1000 alla relativa priorità. Si potrebbe utilizzare una costante differente mantenendo l'algoritmo in grado di fare le stesse scelte?
20. Quando il sistema (UNIX o qualsiasi altro) viene avviato, il loader di bootstrap nel settore 0 del disco carica in primis il programma di avvio che successivamente carica il sistema operativo. Perché è necessario questo passo in più? Di certo sarebbe più semplice avere il loader di bootstrap nel settore 0 che carica direttamente il sistema operativo.
21. Un certo editor ha 100KB di testo di programma, 30KB di dati inizializzati e 50KB di BSS; lo stack inizialmente è di 10KB. Si supponga che partano contemporaneamente tre copie di questo editor; quanta memoria fisica è necessaria se (a) è usato un testo condiviso e (b) nel caso contrario?
22. Nel 4BSD, ogni elemento della core map contiene l'indice dell'elemento successivo nella lista libera, usato quando l'elemento corrente è nella lista libera, il quale occupa 16 bit, e le pagine sono di 1KB. Questi dimensionamenti hanno qualche implicazione per l'ammontare totale di memoria che il 4BSD può supportare? Giustificare la risposta
23. Nel BSD, i segmenti di dati e di stack sono paginati e swapped con una copia tenuta su una pagina speciale del disco, mentre il segmento del testo usa il file binario eseguibile. Perché?
24. Si descriva un modo di usare `mmap` ed i segnali per costruire un meccanismo di comunicazione fra processi.
25. Un file viene mappato utilizzando la seguente chiamata di sistema `mmap`:  
`mmap(65536, 32768, READ, FLAGS, fd, 0)`  
Le pagine sono di 8KB. Quale byte del file viene raggiunto leggendo all'indirizzo di memoria 72000?
26. Dopo una chiamata di sistema come quella del problema precedente, viene fatta la chiamata  
`munmap(65536, 8192)`  
Essa ha successo? Se sì, quali byte del file rimangono mappati? Se no, perché ha fallito?
27. Può succedere che un fault di pagina causi la terminazione del processo? Se sì, si dia un esempio. Se no, perché?
28. È possibile che succeda, con un sistema buddy di gestione della memoria, che due blocchi di memoria libera adiacenti, della stessa dimensione, coesistano senza essere uniti in un unico blocco? Se sì, si dia un esempio di come accade. Se no, si dimostri che non è possibile.
29. Nel testo si è detto che una partizione di paginazione dà prestazioni migliori rispetto ad un file di paginazione. Perché?
30. Si diano due esempi di vantaggio del pathname relativo rispetto a quello assoluto.

31. Le seguenti chiamate di lock sono fatte da un insieme di processi, per ognuna di esse dire che cosa succede. Un processo si ferma se fallisce nel chiedere un lock.
- A vuole un lock condiviso sui byte da 0 a 10.
  - B vuole un lock esclusivo sui byte da 20 a 30.
  - C vuole un lock condiviso sui byte da 8 a 40.
  - A vuole un lock condiviso sui byte da 25 a 35.
  - B vuole un lock esclusivo sul byte 8.
32. Si consideri il file bloccato di Figura 10.27(c). Si supponga che un processo provi a bloccare i byte 10 e 11 e si fermi. Prima che C rilasci il suo lock, un altro processo prova a bloccare i byte 10 e 11, ed anch'esso si ferma. Quali problemi porta una situazione di questo tipo alla semantica del lock? Si propongano e si giustifichino due soluzioni.
33. Si supponga che una chiamata di sistema `lseek` cerchi un offset negativo in un file. Dare due modi possibili per trattarla
34. Si supponga che un file UNIX abbia protezione 755 (ottale). Cosa può fare sul file il proprietario? Il gruppo? Gli altri?
35. Alcuni driver di nastro posseggono blocchi numerati e la capacità di sovrascrivere un particolare blocco in un punto senza disturbare i blocchi prima e dopo. Un sistema del genere potrebbe contenere un file system montato UNIX?
36. In Figura 10.25, sia Fred che Lisa possono accedere al file *x* dalle loro directory dopo aver fatto il collegamento. Questo accesso è da considerarsi completamente simmetrico (nel senso che ognuno di loro può fare con il file tutto ciò che può fare l'altro)?
37. Come si è visto, i pathname assoluti vengono visitati partendo dalla directory radice mentre i pathname relativi partendo dalla directory di lavoro. Suggerire un modo efficiente per implementare questi due tipi di ricerca
38. Quando viene aperto il file `/usr/nst/lavoro/f`, sono necessari molti accessi al disco per leggere gli i-node ed i blocchi di directory. Calcolare il numero di accessi al disco richiesti sapendo che l'i-node della directory radice è sempre in memoria e tutte le directory sono su un solo blocco.
39. Un i-node di UNIX ha 10 indirizzi di disco per i blocchi dati, più quelli per il blocco singolo, doppio e triplo indiretto. Se ognuno di questi contiene 256 indirizzi di disco, qual è la dimensione del file più grande che può essere gestito, con blocchi del disco di 1K?
40. Quando viene letto un i-node da disco durante il processo di apertura di un file, questo viene posto in una tabella di i-node, in memoria. Tale tabella possiede alcuni campi che non sono presenti nel disco; uno di questi è il contatore che mantiene traccia del numero di volte che l'i-node relativo è stato aperto. Per quale ragione tale campo è necessario?
41. Perché la tecnica LRU è adatta a gestire il buffer cache, mentre raramente funziona quando è usata per tener traccia delle pagine in un sistema a memoria virtuale?
42. UNIX possiede una chiamata di sistema, `sync`, che svuota la cache di buffer sul disco. Al momento dell'avvio del sistema, viene lanciato un programma chiamato `update`; il quale ogni 30 secondi chiama `sync` e quindi si riaddormenta per altri 30 secondi. Come mai esiste questo programma?

43. Solitamente, dopo un crash del sistema ed un riavvio, viene eseguito un programma di recupero. Si supponga che questo programma scopra che il contatore di collegamenti ad un i-node disco sia 2 mentre una sola directory si riferisca a quell'i-node. Tale problema può essere risolto? Se sì, come?
44. Fare una stima ragionata di quale chiamata di sistema UNIX è più veloce.
45. È possibile scollegare un file che non è mai stato collegato? Che cosa succede in questo caso?
46. In base alle nozioni date nel capitolo, se un file system Linux ext2 fosse messo in un dischetto da 1,44 MB, quale sarebbe il massimo quantitativo di dati utente (su file) che potrebbero essere immagazzinati sul disco? Si supponga che i blocchi disco siano di 1KB.
47. Sapendo tutti i problemi che uno studente può causare se diventa superuser, come mai questo concetto continua ad esistere?
48. Un professore condivide alcuni file con i propri studenti posizionandoli in una directory accessibile pubblicamente nel sistema UNIX del dipartimento di informatica. Un giorno si rende conto che un uno dei file (che era stato messo il giorno prima) aveva il permesso di scrittura per tutti. Il professore allora cambia i diritti sul file e verifica che il file sia identico alla copia personale. Il giorno successivo, però, scopre che il file è stato cambiato. Come può essere accaduta una cosa del genere? E come può essere impedita?
49. Si scriva una shell minimale che permetta di dare semplici comandi, anche in background.
50. Utilizzando il linguaggio assembler e le chiamate BIOS, si scriva un programma che si auto-avvia da floppy disk su un computer Pentium. Il programma dovrebbe utilizzare le chiamate BIOS per leggere da tastiera e visualizzare i caratteri ricevuti, semplicemente per dimostrare che è in esecuzione.
51. Si scriva un programma che simula un terminale, per connettere due workstation UNIX o Linux via porta seriale. Si usino la chiamate di gestione da terminale POSIX per configurare le porte.

# CASO DI STUDIO 2: WINDOWS 2000

Windows 2000 è un sistema operativo moderno che gira su computer da tavolo di fascia alta e server. In questo capitolo esamineremo vari aspetti del sistema, incominciando da una breve storia e spostandoci, quindi, verso la sua architettura. In seguito osserveremo i processi, la gestione della memoria, l'I/O, il file system ed, infine, gli aspetti di sicurezza; gli aspetti di rete non verranno trattati poiché potrebbero prendere, da soli, un capitolo (se non un libro intero).

### 11.1 Storia di Windows 2000

I sistemi operativi Microsoft per computer da tavolo e portatili possono essere divisi in tre famiglie: MS-DOS, Windows per consumatori (Windows 95/98/Me) e Windows NT. Qui di seguito verrà trattata in dettaglio ognuna delle famiglie.

#### 11.1.1 MS-DOS

Nel 1981 l'IBM, a quei tempi la più grande e potente azienda di computer del mondo, produsse il personal computer IBM basato sull'8088. Esso uscì dotato di un sistema operativo a 16-bit, in modo reale, a utente singolo, a righe di comando, chiamato MS-DOS 1.0. Il sistema operativo era fornito da Microsoft, una piccola azienda appena nata, all'epoca conosciuta soprattutto per il suo interprete BASIC, usato sui sistemi 8080 e Z-80. Questo sistema operativo era composto da 8KB di codice residente in memoria, ed era stato modellato su CP/M, un piccolo sistema operativo per CPU Z-80 e 8080 a 8-bit. Due

anni più tardi uscì un sistema operativo molto più potente, MS-DOS 2.0, che occupava 24 KB, e conteneva un processore di righe di comando (shell) con alcune funzionalità prese in prestito da UNIX.

Quando Intel uscì con il chip 286, IBM costruì un nuovo computer intorno ad esso, il PC/AT, uscito nel 1986. AT stava per Tecnologia Avanzata (Advanced Technology), dato che il 286 girava alla velocità impressionante (per allora) di 8MHz, e poteva indirizzare - con grande difficoltà - tutti i 16MB di RAM. In pratica, la maggior parte dei sistemi aveva, al più, 1MB o 2MB, per il costo elevato della memoria. Il PC/AT era corredata di Microsoft MS-DOS 3.0, di 36KB; nel corso degli anni, MS-DOS continuò ad acquisire nuove funzionalità, ma rimase sempre un sistema operativo a righe di comando.

## 11.1.2 Windows 95/98/Me

Ispirata dall'interfaccia utente grafica dell'Apple Lisa, il predecessore dell'Apple Macintosh, Microsoft decise di dare ad MS-DOS una interfaccia grafica utente (shell) che chiamò Windows. Windows 1.0 uscì nel 1985, e fu un disastro; Windows 2.0, creato per il PC/AT ed uscito nel 1987, non fu per nulla meglio. Finalmente, Windows 3.0 per 386 (uscito nel 1990), ed in particolar modo i suoi diretti successori 3.1 e 3.11, si dimostrarono un enorme successo commerciale. Nessuna di queste prime versioni di Windows era di fatto un vero sistema operativo, ma piuttosto una interfaccia utente grafica costruita su MS-DOS, che continuava a controllare la macchina ed il file system. Tutti i programmi giravano nello stesso spazio di indirizzi, ed un errore in uno solo di questi poteva portare al blocco dell'intero sistema.

L'uscita di Windows 95 nell'Agosto 1995 non riuscì ad eliminare completamente MS-DOS, ma trasportò quasi tutte le funzionalità dalla parte MS-DOS alla parte Windows. Insieme, Windows 95 e il nuovo MS-DOS 7.0, contenevano la maggior parte delle funzionalità di un sistema operativo completo, tra cui la memoria virtuale, la gestione dei processi e la multiprogrammazione. Ad ogni modo, Windows 95 non era completamente un programma a 32-bit: esso conteneva grandi pezzi di vecchio codice assembler a 16-bit (oltre ad un po' di codice a 32-bit) ed utilizzava ancora il file system MS-DOS, con tutte le sue limitazioni. L'unico cambiamento significativo al file system fu l'aggiunta dei nomi lunghi per i file, al posto dei nomi a 8+3 caratteri permessi da MS-DOS.

Anche con l'uscita di Windows 98 nel Giugno del 1998, MS-DOS (con la versione 7.1) era fortemente presente con il suo codice a 16-bit. Anche se nuove funzionalità migrarono dalla parte MS-DOS alla parte Windows, e si adottò una organizzazione del disco più adatta a dischi più spaziosi, alla base di tutto, Windows 98 non era molto diverso da Windows 95. La più grande differenza consisteva nell'interfaccia utente, che integrava in modo più stretto il desktop con Internet. Fu proprio questa integrazione che attirò l'attenzione del Dipartimento di Giustizia Americano, che citò la Microsoft accusandola di avere un monopolio illegale, un'accusa che Microsoft negò con vigore. Nell'Aprile del 2000, la Corte Federale degli Stati Uniti espresse parere favorevole al Governo.

Oltre a contenere un pezzo notevole di codice assembler a 16-bit nel kernel, Windows 98 aveva altri due problemi seri: in primo luogo, anche se era un sistema multiprogrammato, il kernel non era rientrante; quindi, se un processo attivo stava manipolando una qualche struttura dati del kernel e, di colpo, il suo quanto si esauriva ed incominciava un altro processo, il nuovo processo poteva trovare la struttura dati in uno stato inconsistente. Per impedire questo tipo di problema, dopo essere entrati nel kernel, la maggior parte dei processi acquisiva in primo luogo un enorme mutex (mutual exclusion object) sull'intero sistema prima di fare qualsiasi altra cosa. Se da un lato questo stratagemma

eliminava qualsiasi problema di inconsistenza, di fatto eliminava quasi del tutto la multiprogrammazione; sovente i processi venivano forzati ad attendere che uscissero dal kernel altri processi, ad essi per nulla correlati, prima di potervi entrare.

In secondo luogo, ogni processo di Windows 98 aveva uno spazio di indirizzi virtuale di 4GB; di questi, 2GB erano completamente riservati al processo, il successivo gigabyte era condiviso (scrivibile) tra tutti i processi nel sistema, ed il megabyte finale era ugualmente condiviso tra tutti i processi, in modo da permettere a tutti di accedere al vettore di interruzione di MS-DOS. Questa funzione di condivisione era pesantemente utilizzata dalla maggior parte delle applicazioni di Windows 98, di conseguenza, un problema in un programma poteva eliminare strutture dati chiave per altri programmi ad esso non correlati, obbligandoli all'arresto immediato. Peggio ancora, l'ultimo gigabyte era condiviso (scrivibile) con il kernel, e conteneva alcune strutture dati critiche del kernel: un qualsiasi programma "mascalzone" che scrivesse dati casuali sopra queste strutture avrebbe portato l'intero sistema al black. La soluzione ovvia a questo problema, vale a dire quella di non mettere le strutture dati di kernel nello spazio utente, non era attuabile, perché questa funzionalità era essenziale per far funzionare i vecchi programmi MS-DOS in Windows 98.

Nell'anno del millennio, 2000, Microsoft fece uscire una piccola revisione di Windows 98, chiamata Windows Me (Windows Millennium Edition); malgrado questa versione risolvesse alcuni problemi ed aggiungesse un po' di funzionalità, di base era ancora Windows 98. Le nuove funzionalità comprendevano modi migliori per catalogare e condividere immagini, musica e filmati, maggior supporto per la gestione della rete e per giochi multiutente, e più funzionalità correlate ad Internet, come, ad esempio, il supporto per lo scambio di messaggi istantanei e le connessioni a banda larga (modem via cavo e ADSL). Una nuova funzionalità importante era la possibilità di riportare il sistema alla configurazione precedente nel caso di errore nella configurazione: se l'utente riconfigurava il sistema (ad esempio cambiando la risoluzione dello schermo da 640x480 a 1024x768) ed esso non funzionava più, questa funzionalità permette di ritornare alla configurazione precedente, che invece funzionava.

### 11.1.3 Windows NT

Verso la fine degli anni '80, la Microsoft si accorse che, probabilmente, costruire un sistema operativo moderno a 32-bit al di sopra del debole MS-DOS a 16-bit, non era la soluzione migliore. Per questa ragione reclutò David Cutler, uno dei principali progettisti del sistema operativo VMS della DEC, e gli diede il compito di organizzare un team di lavoro per produrre un sistema operativo Windows compatibile a 32-bit, completamente nuovo e senza dipendenze dal vecchio. Questo nuovo sistema, più tardi chiamato Windows NT (Nuova Tecnologia, New Technology), venne studiato per rispondere alle esigenze delle applicazioni critiche delle aziende, ma anche per i singoli utenti. A quei tempi i mainframe dominavano ancora il mercato degli affari mondiale, e, per questo, l'obiettivo di progettare un sistema operativo, nell'ipotesi che tutte le società usassero i personal computer per le loro attività principali, sembrava alquanto visionario, ma si rivelò nel tempo una scelta più che ottimale. Funzionalità come sicurezza e alta affidabilità, che mancavano visibilmente nelle versioni di Windows basate su MS-DOS, divennero obiettivi principali del progetto di Windows NT. Le conoscenze di Cutler sul VMS si scorgono chiaramente in vari aspetti, rendendo la somiglianza tra il progetto di NT e quello di VMS abbastanza evidente.

Il progetto ebbe successo e la prima versione del sistema, chiamata Windows NT 3.1 uscì nel 1993. Non a caso, si scelse il numero di questa prima versione uguale a quello

della popolare Windows 3.1 a 16-bit; Microsoft si aspettava che il nuovo sistema, notevolmente superiore, sostituisse con rapidità il vecchio Windows 3.1.

Con sorpresa si accorse che quasi tutti gli utenti preferivano rimanere al vecchio sistema a 16-bit che ben conoscevano, piuttosto che passare ad un moderno sistema operativo a 32-bit che non conoscevano affatto, per quanto migliore potesse essere. In più, NT richiedeva molta più memoria di Windows 3.1, e non esistevano programmi a 32-bit da far girare su NT, quindi, perché cambiare? Il fallimento di Windows NT 3.1 sul mercato fu la ragione principale che spinse Microsoft a progettare Windows 95, un sistema ibrido a 32-bit, basato però su Windows 3.1. La continua resistenza degli utenti a NT causò, in seguito, la produzione di Windows 98 e, infine, di Windows Me; e tutti annunciarono di essere l'ultimissima versione di un sistema operativo basato su MS-DOS.

Sebbene praticamente tutti i clienti e le aziende ignorassero il sistema NT 3.1, esso acquistò un piccolo spazio di mercato nell'ambiente dei server; nel 1994 e 1995 uscirono alcune versioni 3.x di NT, che lentamente incominciarono ad essere adottate anche da alcuni utenti finali.

L'aggiornamento più evidente dell'architettura NT si ebbe nel 1996, con NT 4.0: questo sistema possedeva la potenza, la sicurezza e l'affidabilità dei moderni sistemi operativi, ma anche una interfaccia identica a quella, ormai popolare, di Windows 95. Tale compatibilità rendeva più semplice la migrazione degli utenti da Windows 95 a NT 4.0 e molti utenti la concretizzarono; alcune delle differenze tra Windows 95/98 e Windows NT sono riassunte in Figura 11.1.

Fin dall'inizio NT è stato studiato per essere portatile e, per questo, è stato scritto quasi interamente in C, con una piccola quantità di codice assembler dedicato principalmente alle funzioni di basso livello, tra cui la gestione delle interruzioni. La versione iniziale si componeva di 3,1 milioni di righe di codice C per il sistema operativo, le librerie e i sotto-sistemi di ambiente (discorsi più avanti). Quando uscì NT 4.0, il codice era cresciuto a 16 milioni di righe di codice, ancora in massima parte in C, ma anche con un piccolo pezzo di codice C++, per la parte di interfaccia utente. Fino a questo momento il sistema era

| Caratteristica                                            | Windows 95/98 | Windows 2000            |
|-----------------------------------------------------------|---------------|-------------------------|
| Un sistema completo 32 bit?                               | No            | Sì                      |
| Sicurezza?                                                | No            | Sì                      |
| Mappe di file protetti?                                   | No            | Sì                      |
| Spazio degli indirizzi privato per ogni programma MS-DOS? | No            | Sì                      |
| Unicode?                                                  | No            | Sì                      |
| Gira su...                                                | Intel 80x86   | 80x86, Alpha, MIPS, ... |
| Supporto di multiprocessori?                              | No            | Sì                      |
| Codice rientrante nel sistema operativo?                  | No            | Sì                      |
| Plug and play?                                            | Sì            | No                      |
| Gestione dell'alimentazione?                              | Sì            | No                      |
| File system FAT-32?                                       | Sì            | Opzionale               |
| File system NTFS?                                         | No            | Sì                      |
| API Win32?                                                | Sì            | Sì                      |
| Può eseguire tutti i vecchi programmi MS-DOS?             | Sì            | No                      |
| Alcuni dati del sistema scrivibili da parte di utenti?    | Sì            | No                      |

Figura 11.1 Alcune differenze tra Windows 98 e Windows NT.

altamente portabile, con versioni che giravano su Pentium, Alpha, MIPS e PowerPC, ed altre CPU; alcune di queste scomparse da allora. La storia di come fu sviluppato il sistema NT è raccontata nel libro *Showstopper* (Zachary, 1994); nel libro vi sono inoltre molti dettagli sulle persone coinvolte nello sviluppo.

### 11.1.4 Windows 2000

La versione NT successiva a NT 4.0 doveva essere chiamata inizialmente NT 5.0. Ad ogni modo, nel 1999, Microsoft cambiò il nome in Windows 2000, fondamentalmente per adottare un nome neutro, in modo che sia gli utenti di Windows 98 sia quelli di NT 4.0 potessero vederlo come una logica evoluzione. Se questo progetto andrà in porto, Microsoft si troverà con un solo sistema operativo, costruito con tecnologia a 32-bit, e con la ben nota interfaccia utente di Windows 98.

Poiché Windows 2000 è NT 5.0, esso eredita molte proprietà di NT 4.0: prima di tutto è un vero sistema in multiprogrammazione a 32-bit (presto a 64-bit), con processi individualmente protetti. Ogni processo possiede uno spazio degli indirizzi virtuale di 32-bit (presto a 64-bit) paginato a richiesta; il sistema operativo viene eseguito in modo kernel, mentre i processi girano in modo utente, e ciò garantisce una protezione completa (senza i difetti di protezione di Windows 98); i processi possono avere uno o più thread, che sono visibili e schedulati dal sistema operativo. Windows 2000 ha la classificazione di sicurezza C2 del Dipartimento della Difesa, per ogni file, directory e ogni altro oggetto condivisibile (purché il floppy sia rimosso e la rete non connessa). Infine, possiede un supporto completo per l'esecuzione su multiprocessori simmetrici fino a 32 CPU.

Il fatto che Windows 2000 sia in realtà NT 5.0 risulta evidente in molti contesti; per esempio, la directory di sistema è chiamata \winnt ed il kernel binario del sistema operativo (che si trova in \winnt\system32) è detto ntoskrnl.exe. Selezionando questo file con il bottone destro del mouse per esaminarne le proprietà, si scopre che il numero di versione è 5.xxx.yyy.zzz dove 5 sta per NT 5, xxx per il numero di versione, yyy è il numero di compilazione, zzz la variante minore: inoltre, molti file in \winnt e nelle sottodirectory hanno il prefisso nt, come ad esempio ntadm, l'emulatore virtuale di MS-DOS sotto NT.

Windows 2000 è più di un NT 4.0 migliorato con interfaccia utente Windows 98: in primo luogo, esso contiene delle funzionalità riscontrate in precedenza solo in Windows 98, tra cui il supporto completo per dispositivi plug-and-play, il bus USB, IEEE 1394 (FireWire), IrDA (il collegamento a infrarossi tra computer portatili e stampanti) e la gestione dell'alimentazione. In più, sono state aggiunte alcune funzionalità non presenti in nessun altro sistema operativo Microsoft, tra cui il servizio di active directory, la sicurezza usando Kerberos, il supporto per smart card, gli strumenti per il monitoraggio del sistema, una migliore integrazione tra computer da tavolo e portatili, un'infrastruttura per la gestione del sistema e gli oggetti di lavoro. È stato anche esteso il file system principale, NTFS, per supportare per esempio i file crittati, le quote, i collegamenti tra file, i volumi montati e l'indicizzazione di contenuto. Un'altra funzionalità nuova dell'NTFS è il deposito di istanze singole, una sorta di collegamento con copia in scrittura (copy-on-write), per cui due utenti possono condividere un file collegato, fino a quando uno dei due non vi scrive sopra, ed in quel momento, si crea automaticamente una copia per l'altro utente.

Un altro miglioramento di rilievo è l'internazionalizzazione: NT 4.0 nasci in versioni separate per lingue diverse con stringhe di testo inglobate nel codice, e spesso succedeva che, installando un pacchetto di software Inglese in un computer Tedesco, altri programmi smetessero di utilizzare messaggi in tedesco ed incominciassero ad usare l'inglese, dato che certi file venivano sovrascritti con la nuova installazione. Questo problema è sta-

to eliminato: Windows 2000 ha un unico binario che gira in ogni parte del mondo, ed una installazione, o anche un singolo utente, può scegliere la lingua da utilizzare a tempo di esecuzione, perché tutti gli oggetti di menu, le stringhe di dialogo, i messaggi di errore e le altre stringhe di testo sono state tolte dal sistema operativo e messe in directory separate, una per lingua installata. Come tutte le versioni precedenti di NT, Windows 2000 utilizza Unicode in tutto il sistema, per supportare linguaggi che non usano l'alfabeto latino, come il Russo, il Greco, l'Ebreo ed il Giapponese.

Una cosa che Windows 2000 non ha è MS-DOS: semplicemente, non è presente in nessuna forma (non lo era neanche in NT). Vi è una interfaccia a righe di comando, ma è un nuovo programma a 32-bit che contiene le vecchie funzionalità MS-DOS, oltre a parecchie altre nuove.

Nonostante le molte caratteristiche di portabilità del codice, l'hardware, le lingue eccetera, in una cosa Windows 2000 può considerarsi meno portabile di NT 4.0: esso può girare solo su due piattaforme, il Pentium e l'Intel IA-64. In origine, NT supportava altre piattaforme come PowerPC, MIPS, Alpha ma, negli anni, Microsoft le ha abbandonate una ad una per ragioni di tipo commerciale.

Come nel caso delle versioni precedenti di NT, Windows 2000 è uscito come prodotto a diversi livelli: Professionale, Server, Server Avanzato, Server DataCenter. Le differenze tra tutte queste versioni sono comunque di basso rilievo, visto l'uso dello stesso binario per tutte. Quando il sistema viene installato, il tipo di prodotto viene registrato in un database interno (il registro); e in fase di caricamento, il sistema operativo controlla il registro per vedere di che versione si tratta. Le differenze sono mostrate in Figura 11.2.

Come mostrato in figura, le differenze sono: la massima memoria supportata, il massimo numero di CPU (per la configurazione multiprocessore), il massimo numero di client che possono essere serviti. Il numero di processori si riferisce alla capacità di Windows 2000 di far apparire due o quattro macchine come un singolo server, per il mondo esterno, il che rappresenta, ad esempio, una possibilità interessante per i web server. Infine, i parametri per difetto vengono tarati diversamente per la versione Professionale, per favorire i programmi interattivi rispetto al lavoro in batch, sebbene, volendo, questi parametri possano essere facilmente modificati. L'ultima differenza sta nel fatto che viene fornito del software in più sulle versioni server, e qualche strumento supplementare nella versione DataCenter, dedicato alla gestione di grandi carichi di lavoro.

La ragione per cui si hanno più versioni è semplicemente una ragione di vendita: in questo modo Microsoft può caricare il prezzo per le grandi aziende, e diminuirlo per i semplici utenti, vendendo in pratica lo stesso prodotto. Ad ogni modo, questa idea non è di certo nuova e di proprietà Microsoft; per anni, le compagnie aeree hanno aumentato il prezzo dei biglietti per i passeggeri non solo per la Business Class, ma anche per la Classe Bestiame, se un passeggero voleva permettersi il lusso di acquistare il biglietto il giorno prima di partire, invece del inese prima.

Tecnicamente parlando, le differenze tra le varie versioni stanno in pochi posti del codice, grazie a due variabili lette dal registro, *ProductType* e *ProductSuite*: in base al loro

| Versione          | RAM max. | Numero CPU | Client max. | Dim. cluster | Ottimizzata per   |
|-------------------|----------|------------|-------------|--------------|-------------------|
| Professional      | 4 GB     | 2          | 10          | 0            | Tempo di risposta |
| Server            | 4 GB     | 4          | Illimitato  | 0            | Throughput        |
| Server Avanzato   | 8 GB     | 8          | Illimitato  | 2            | Throughput        |
| Server Datacenter | 64 GB    | 32         | Illimitato  | 4            | Throughput        |

Figura 11.2 Le diverse versioni di Windows 2000.

| Anno | AT&T      | BSD           | MINIX   | Linux    | Solaris  | Win NT   |
|------|-----------|---------------|---------|----------|----------|----------|
| 1976 | V6 9K     |               |         |          |          |          |
| 1979 | V7 21K    |               |         |          |          |          |
| 1980 |           | 4.1 38K       |         |          |          |          |
| 1982 | Sys V 58K |               |         |          |          |          |
| 1984 |           | 4.2 98K       |         |          |          |          |
| 1986 |           | 4.3 179K      |         |          |          |          |
| 1987 | SVR3 92K  |               | 1.0 13K |          |          |          |
| 1989 | SVR4 280K |               |         |          |          |          |
| 1991 |           |               |         | 0.01 10K |          |          |
| 1993 |           | Free 1.0 235K |         |          | 5.3 850K | 3.1 6M   |
| 1994 |           | 4.4 Lite 743K |         | 1.0 165K |          | 3.5 10M  |
| 1996 |           |               |         | 2.0 470K |          | 4.0 16M  |
| 1997 |           |               | 2.0 62K |          | 5.6 1.4M |          |
| 1999 |           |               |         | 2.2 1M   |          |          |
| 2000 |           | Free 4.0 1.4M |         |          | 5.8 2M   | 2000 29M |

**Figura 11.3** Un confronto di alcune dimensioni di sistemi operativi. La prima stringa in ogni casella è la versione; la seconda è la dimensione rispetto al numero di righe di codice dove K = 1000 e M = 1 000 000. I confronti all'interno di una colonna hanno un senso. Quelli fra le colonne no, come discusso all'interno del paragrafo.

valore, si esegue del codice leggermente diverso. Cambiare i valori di queste variabili rappresenta una violazione di licenza; inoltre, il sistema esegue una trap ad ogni tentativo di cambiamento, e registra il tentativo in modo indelebile, per poterlo rilevare in un secondo momento.

Oltre al sistema operativo di base, Microsoft ha sviluppato parecchi strumenti per gli utenti avanzati, tra cui gli Strumenti di supporto (Support Tools), il Pacchetto di sviluppo software (Software Development Kit), Pacchetto di sviluppo driver (Driver Development Kit), ed il Pacchetto di risorse (Resource Kit); in essi vi è un vastissimo numero programmi di utilità e strumenti per interagire e controllare il sistema. Gli strumenti di supporto si possono trovare nel CD-ROM di Windows 2000, nella directory \support\tools; la procedura standard di installazione Windows, di fatto, non li installa ma vi è un file setup.exe, in questa directory, che lo fa. L'SDK ed il DDK sono disponibili per i programmati all'indirizzo [msdn.microsoft.com](http://msdn.microsoft.com), ma il Resource Kit è un prodotto Microsoft a parte. Vi sono poi molti altri strumenti di altri produttori per indagare all'interno di Windows 2000, tra cui un insieme gratuito di strumenti, disponibile in rete al sito [www.sysinternals.com](http://www.sysinternals.com); alcuni di questi riescono a fornire informazioni più dettagliate di quanto non fanno gli strumenti Microsoft.

Windows 2000 è un sistema immensamente complesso, composto da più di 29 milioni di righe di codice C; se si stampassero 50 righe di codice per pagina ed ogni libro potesse avere 1000 pagine, il codice completo riempirebbe 580 volumi, che in edizione economica occuperebbero 23 metri di mensola. Sistemati in librerie lunghe un metro con 6 mensole per ognuna, l'insieme occuperebbe un muro lungo 4 metri.

Giusto per divertimento, in Figura 11.3 si fa un confronto tra alcuni sistemi operativi, in termini di dimensione di codice sorgente; in ogni modo questa tabella dovrebbe essere presa con un grano (o meglio, una tonnellata) di sale, dal momento che ciò che componete il sistema operativo è diverso per diversi sistemi. Per esempio, tutto il sistema a

finestre e la GUI fanno parte del kernel in Windows, ma non stanno nelle versioni di UNIX, in cui sono semplicemente processi utente. Considerando X-Windows, si possono aggiungere altri 15 milioni di righe di codice a tutte le versioni di UNIX, e non si considera ancora il codice GUI (Motif, GNOME, eccetera) che, nel mondo UNIX, non fa parte del sistema operativo. In più, alcuni sistemi contengono codice per più architetture (ad esempio, cinque per 4.4 BSD e nove per Linux), aggiungendo da 10000 a 50000 righe di codice per ogni architettura. La ragione per cui Free BSD 1.0 ha solo 235000 righe di codice rispetto a 4BSDLite (con 743000 righe), da cui è stato ricavato, è che da Free BSD è stato eliminato il supporto di tutte le architetture obsolete (ad esempio VAX).

Anche il numero dei file system, dei driver di dispositivi, e delle librerie fornite variano notevolmente da sistema a sistema; in più, Windows contiene una grande quantità di codice di test che UNIX non contiene, come pure un certo numero di programmi di utilità e il supporto per numerose lingue, oltre all'Inglese. Infine, le misurazioni sono state fatte da persone diverse, il che introduce una certa varianza (ad esempio, i makefile, gli header, i file di configurazione e la documentazione hanno influito? E di quanto?). Questo non è come confrontare le mele con le arance; è come confrontare mele con telefoni; comunque i conti, all'interno della stessa famiglia di sistemi, sono stati fatti dalla stessa organizzazione e quindi sono abbastanza significativi.

Nonostante tutte queste avvertenze, si possono trarre chiaramente due conclusioni:

1. Il dilatarsi dei sistemi sembra essere inevitabile come la morte e le tasse.
2. Windows è molto più grande di UNIX.

Il problema se sia meglio un codice piccolo o grande è una controversia bollente da sempre; le argomentazioni a favore del piccolo stanno nel fatto che piccole dimensioni ed una concezione semplice e lineare producono un sistema maneggevole e affidabile, che gli utenti possono facilmente comprendere. Le argomentazioni a favore di grande stanno nel fatto che molti utenti vogliono tante funzionalità; ma in tutti i casi dovrebbe essere chiaro che tutti gli studenti che progettano di scrivere un sistema operativo completo, allo stato dell'arte, da zero, hanno un bel lavoro che li attende.

Sebbene Windows 2000 sia ormai il campione mondiale in termini di massa pura, è in continua crescita, con problemi che vengono risolti e nuove funzionalità che vengono aggiunte. Vale la pena soffermarsi sul modo in cui Microsoft gestisce lo sviluppo. Centinaia di programmatore lavorano su vari aspetti di Windows 2000 tutti i giorni, ed ogni volta che si completa un pezzo di codice, il programmatore lo invia elettronicamente al team di costruzione; alle 18 di ogni giorno, la porta viene chiusa ed il sistema viene ricostruito (ovvero ricompilato e ricollegato). Ogni ricostruzione ha un numero di sequenza unico, che può essere visto esaminando il numero di release di *ntoskrnl.exe* (la prima release pubblica di Windows 2000 aveva numero 2195).

Il nuovo sistema operativo viene distribuito in modo automatico elettronicamente a migliaia di macchine nel campus della Microsoft di Redmond, WA, dove viene testato intensamente per tutta la notte. La mattina presto del giorno successivo, i gruppi responsabili ricevono i risultati di tutti i test, in modo che essi possano vedere se il loro nuovo codice ha funzionato bene. Quindi ogni gruppo decide su quale codice vuole lavorare, e per tutto il giorno il gruppo di programmatore lavora sul codice assegnato, fino alle 18, quando il ciclo di test ricomincia.

## 11.2 Programmare in Windows 2000

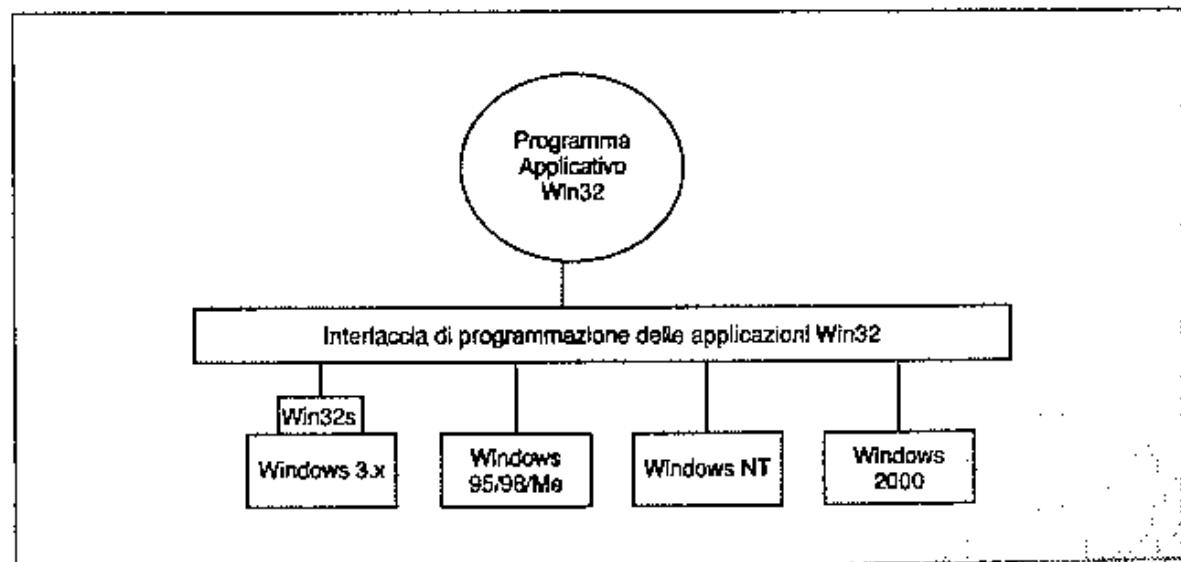
Ora è tempo di cominciare il nostro studio tecnico di Windows 2000. Prima di entrare nei dettagli della struttura interna, dovremo esaminare l'interfaccia di programmazione ed il registro, una piccola base dati in memoria.

### 11.2.1 L'interfaccia di programmazione delle applicazioni (API) Win32

Come tutti gli altri sistemi operativi, Windows 2000 ha un insieme di chiamate di sistema che possono essere attivate; tuttavia, Microsoft non ha mai reso pubblica tale lista, che viene cambiata da una release all'altra. Per contro, ciò che fa Microsoft è definire un insieme di chiamate di funzione chiamate Win32 API (Win32 Application Programming Interface - Interfaccia di programmazione delle applicazioni), che sono rese note al pubblico e completamente documentate; esse sono procedure di libreria, che effettuano chiamate di sistema per fare il lavoro richiesto, o, in alcuni casi, fanno il lavoro esse stesse nello spazio utente. Le chiamate Win32 API esistenti non cambiano con le nuove release di Windows, tuttavia vengono aggiunte di frequente nuove chiamate API.

I programmi binari per l'Intel x86, che aderiscono esattamente all'interfaccia Win32 API, gireranno senza modifiche in tutte le versioni di Windows successive a Windows 95. Come mostra la Figura 11.4, per Windows 3.x è necessaria una libreria aggiuntiva per avere la corrispondenza tra il sottoinsieme di chiamate di Win32 API ed il sistema operativo a 16-bit; per gli altri sistemi non è necessario alcun adattamento. È opportuno notare che Windows 2000 aggiunge molte nuove funzionalità a Win32, quindi ha nuove chiamate API non comprese nelle vecchie versioni di Win32, che non saranno compatibili con le vecchie versioni di Windows.

La filosofia di Win32 API è completamente differente da UNIX: in quest'ultimo, le chiamate di sistema erano tutte pubbliche e formavano un'interfaccia minimale: rimuovendone anche una sola, si riducevano le funzionalità del sistema operativo. La filosofia



**Figura 11.4** L'API Win32 permette ai programmi di girare su praticamente tutte le versioni di Windows.

di Win32 è di fornire un'interfaccia onnicomprensiva, spesso con tre o quattro modi per fare la stessa cosa, e che comprende molte funzioni (i.e. procedure) che chiaramente potrebbero non essere (e non sono) chiamate di sistema, come ad esempio una chiamata API per copiare un intero file.

Molte chiamate Win32 API creano oggetti del kernel di diverso tipo, come file, processi, thread, pipe, e così via; ogni chiamata che crea un oggetto rende al chiamante un risultato chiamato **handle** (**gestore**), che può essere utilizzato in seguito per effettuare operazioni sull'oggetto. I gestori sono specifici del processo che crea l'oggetto riferito dal gestore stesso, e non possono essere passati direttamente ad altri processi ed utilizzati da questi ultimi (proprio come i descrittori di file UNIX non possono essere passati ad altri processi e da loro utilizzati). Tuttavia, in certi casi, è possibile duplicare un gestore e passarlo ad altri processi in modo protetto, permettendo a questi ultimi un accesso controllato ad oggetti appartenenti ad altri processi. Ogni oggetto ha associato anche un descrittore di sicurezza, che dice in dettaglio chi può effettuare quali tipi d'operazioni sull'oggetto, e chi no.

Non tutte le strutture dati create dal sistema sono oggetti, e non tutti gli oggetti sono oggetti del kernel; gli unici che sono veri oggetti del kernel sono quelli che devono essere nominati, protetti, o condivisi in qualche modo. Ogni oggetto del kernel ha una definizione di tipo a livello di sistema, ha operazioni ben determinate definite su esso, ed occupa spazio nella memoria del kernel. Anche se gli utenti possono effettuare delle operazioni (facendo chiamate Win32), non possono accedere ai dati direttamente.

Anche il sistema operativo può creare ed utilizzare oggetti, e lo fa pesantemente; la maggior parte di questi oggetti è creata per permettere ad una componente del sistema di salvare informazioni per un lungo periodo di tempo, o per passare qualche struttura dati ad un'altra componente. Per esempio, quando si carica un driver di periferica, si crea un oggetto che contiene le sue proprietà e i puntatori alle sue funzioni; dopodiché, in tutto il sistema operativo, il driver è indirizzato tramite l'oggetto corrispondente.

Si dice talvolta che Windows 2000 è orientato agli oggetti perché l'unico modo di manipolare oggetti è chiamare operazioni sui gestori tramite Win32 API; tuttavia, gli mancano le proprietà basilari dei sistemi orientati ad oggetti quali l'ereditarietà e il polimorfismo.

Le chiamate Win32 API coprono ogni area concepibile con la quale un sistema operativo potrebbe avere a che fare, e anche un po' che non ha nulla a che vedere. Naturalmente, ci sono chiamate per creare e gestire processi e thread; ci sono anche molte chiamate relative a comunicazioni fra processi (in realtà fra thread), come: creazione, distruzione e utilizzo di mutex, semafori, eventi ed altri oggetti di comunicazione tra processi.

Sebbene la maggior parte del sistema di gestione della memoria sia invisibile ai programmati (fondamentalmente, è solo paginazione su richiesta), risulta visibile una funzionalità importante: la possibilità di mappare un file in una regione della memoria virtuale di un processo. Questo permette ai processi di leggere e scrivere parti di file come se fossero porzioni di memoria.

Un'area importante per molti programmi è l'I/O su file: nella visione di Win32, un file è solo una sequenza lineare di byte. Win32 fornisce oltre 60 chiamate per creare e distruggere file e directory, aprire e chiudere file, leggerli e scriverli, richiedere e definire attributi di file, ed altro ancora.

Un'altra area per la quale Win32 mette a disposizione delle chiamate è la sicurezza: ogni processo ha un identificatore che dice chi è, ed ogni oggetto può avere una lista di controllo degli accessi, che fornisce informazioni dettagliate su quali utenti possono accedere e quali operazioni essi possono fare. Quest'approccio fornisce un livello di sicurezza con granularità molto fine, che permette di definire esattamente i privilegi di accesso ad ogni oggetto per ogni singolo individuo.

I processi, i thread, la sincronizzazione, la gestione della memoria, l'I/O dei file e le chiamate di sistema per la sicurezza non sono nulla di nuovo; altri sistemi operativi possiedono questo tipo di chiamate di sistema, sebbene non in quantità elevata come in Win32. Ciò che realmente distingue Win32 sono le migliaia e migliaia di chiamate per l'interfaccia grafica: vi sono chiamate per creare, distruggere, gestire ed usare finestre, menu, barre degli strumenti, barre di stato, barre di scorrimento, finestre di dialogo, icone e molti altre entità che appaiono sullo schermo. Vi sono chiamate per disegnare figure geometriche, riempirle, gestire la tavolozza dei colori utilizzata, gestire i caratteri di stampa e porre le icone sullo schermo. Infine, vi sono chiamate per la tastiera, il mouse ed altri dispositivi di input; così come per audio, stampanti ed altri dispositivi di output. In breve, la Win32 API (specialmente la parte GUI) è immensa, e non è possibile descriverne i dettagli in questo capitolo, quindi non lo faremo; i lettori interessati potranno consultare uno dei molti libri esistenti su Win32 (ad esempio Petzold, 1999; Simon, 1997; Rector e Newcomer 1997).

Sebbene la Win32 API sia disponibile su Windows 98 (come pure sul sistema operativo per palmari Windows CE), non tutte le versioni di Windows implementano tutte le chiamate e, talvolta, vi sono anche piccole differenze. Per esempio, Windows 98 non ha alcun livello di sicurezza, quindi sotto Windows 98 le chiamate API relative alla sicurezza restituiscono solo codici di errore; inoltre i nomi di file in Windows 2000 utilizzano l'insieme di caratteri Unicode, che non è disponibile in Windows 98, e i nomi di file di Windows 98 non considerano le maiuscole/minuscole, mentre i nomi di file Windows 2000 lo fanno (sebbene qualche tipo di ricerca sui nomi di file non consideri tali differenze). Vi sono anche differenze nei parametri passati ad alcune chiamate di funzioni API; sotto Windows 2000, per esempio, tutte le coordinate dello schermo nelle funzioni grafiche sono numeri veramente a 32 bit; sotto Windows 98 sono utilizzati solo i 16 bit meno significativi, poiché la maggior parte del sottosistema grafico è ancora codificata a 16 bit. L'esistenza della Win32 API su molti sistemi operativi differenti rende più semplice la migrazione di programmi fra essi, ma poiché esistono piccole varianti, ci vuole un po' di attenzione per arrivare alla portabilità.

## 11.2.2 Il registro

Windows deve memorizzare un'enorme quantità di informazioni relative all'hardware, al software, e agli utenti. In Windows 3.x queste informazioni erano immagazzinate in centinaia di file .ini (inizializzazione) sparsi in tutto il disco; da Windows 95 in poi, più o meno tutte le informazioni necessarie per l'avvio e la configurazione del sistema, e per adattarlo all'utente corrente, furono spostate in un grande database centrale chiamato **registro** (registry). In questa sezione daremo una visione generale del registro di Windows 2000.

Tanto per cominciare vale la pena notare che, sebbene molte parti di Windows 2000 siano molto complicate e disordinate, il registro è una delle peggiori, e la nomenclatura crittografica non migliora la situazione; fortunatamente, sono stati scritti interi libri per descriverlo (Born 1998; Hipson 2000 e Ivens 1998). Detto questo, l'idea che risiede dietro al registro è molto semplice: il registro è una collezione di directory, ognuna delle quali contiene o sottodirectory oppure campi dato; in pratica è simile ad un file system per piccoli file, con directory e campi dato (i file).

La confusione comincia dal fatto che Microsoft chiama una directory con il nome di **chiave** (key), mentre di fatto non lo è; in più tutte le directory di livello più alto cominciano con la stringa **HKEY**, che vuol dire gestore della chiave (Handle to Key); le sottodirectory tendono ad avere dei nomi migliori, sebbene questo non accada sempre.

Alla base della gerarchia vi sono i campi dato, chiamati **valori**, che contengono le informazioni. Ogni valore ha 3 parti: un nome, un tipo ed il dato. Il nome è semplicemente una stringa Unicode, spesso default se la directory contiene solo un valore. Il tipo è uno degli undici tipi standard, di cui i più comuni sono: una stringa Unicode, una lista di stringhe Unicode, un intero a 32 bit, un numero binario di lunghezza arbitraria, un collegamento simbolico ad una directory o ad un qualsiasi altro campo dato nel registro. I nomi simbolici sono completamente analoghi ai collegamenti (link) simbolici nei file system o nelle scocciatoie (shortcut) del desktop di Windows: essi permettono un punto di accesso ad un'altra entità o directory. Un link simbolico può anche essere usato come chiave, nel senso che qualcosa che sembra una directory non è altro che un puntatore ad una directory differente.

A livello più alto, il registro di Windows ha sei chiavi, chiamate **chiavi radice**, come mostrato in Figura 11.5, dove sono mostrate altre importanti sottochiavi. Per vedere la lista del tuo sistema operativo, utilizza un semplice editor di registro, sia esso *regedit* o *regedit32* che, sfortunatamente, visualizzano informazioni differenti ed utilizzano diversi formati; entrambi consentono di cambiare i valori di registro. Gli utenti inesperti non dovrebbero provare a cambiare i valori dei campi su nessun sistema se vogliono poter riavviare il computer, però limitarsi a guardare è sicuro. Uomo avvisato, mezzo salvato.

La prima chiave (o directory), **HKEY\_LOCAL\_MACHINE**, è probabilmente la chiave più importante poiché contiene tutte le informazioni sul sistema locale; essa contiene cinque sottochiavi (ovvero sottodirectory). La sottochiave **HARDWARE** contiene molte sot-

| Chiave                | Descrizione                                                                 |
|-----------------------|-----------------------------------------------------------------------------|
| HKEY_LOCAL_MACHINE    | Proprietà di hardware e software                                            |
| HARDWARE              | Descrizione dell'hardware e mappatura dell'hardware ai driver               |
| SAM                   | Informazioni per gli utenti sui livelli di sicurezza e sulla contabilità    |
| SECURITY              | Politiche di sicurezza generali del sistema                                 |
| SOFTWARE              | Informazioni generali sui programmi applicativi installati                  |
| SYSTEM                | Informazioni per l'avvio del sistema                                        |
| HKEY_USERS            | Informazioni su utenti: una sottochiave per utente                          |
| USER-AST-ID           | Profilo di utente AST                                                       |
| AppEvents             | Quale suono produrre quando (arriva e-mail/fax, errore, eccetera...)        |
| Console               | Impostazioni del prompt di comandi (colore, carattere, storia, eccetera...) |
| Control Panel         | Caratteristiche del desktop, salva-schermo, sensibilità del mouse           |
| Environment           | Variabili di ambiente                                                       |
| Keyboard Layout       | Quale tastiera: americana 102 tasti, AZERTY, Dvorak, ...                    |
| Printers              | Informazioni sulle stampanti installate                                     |
| Software              | Preferenze degli utenti su programmi Microsoft e non Microsoft.             |
| HKEY_PERFORMANCE_DATA | Centinaia di contatori per controllare le prestazioni del sistema           |
| HKEY_CLASSES_ROOT     | Link verso HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES                              |
| HKEY_CURRENT_CONFIG   | Link verso i profili hardware correnti                                      |
| HKEY_CURRENT_USER     | Link verso il profilo utente corrente                                       |

**Figura 11.5** Le chiavi di radice, del registro e alcune sottochiavi. Le lettere maiuscole non hanno particolare significato, ma seguono lo schema Microsoft, in questo caso.

tochiavi che restituiscono tutte le informazioni sull'hardware e su quali driver controllano quali pezzi di hardware; tali informazioni sono costruite in linea dal gestore del plug-and-play non appena il sistema viene avviato, infatti, diversamente dalle altre chiavi, essa non viene immagazzinata su disco.

La sottochiave SAM (gestore di sicurezza degli account - Security Account Manager) contiene i nomi degli utenti, i gruppi, le password, ed altre informazioni sugli account e sulla sicurezza necessarie affinché un utente possa accedere al sistema. La sottochiave SECURITY contiene informazioni generali sulle politiche di sicurezza, come la lunghezza minima delle password, quanti tentativi di accesso falliti sono consentiti, eccetera.

La sottochiave SOFTWARE è la sottochiave dove i fornitori di software memorizzano le informazioni; per esempio, se un utente ha installato *Adobe Acrobat*, *Photoshop* e *Premiere*, esisterà la sottochiave *Adobe* e poi ulteriori sottochiavi di questa per *Acrobat*, *Photoshop*, *Premiere* e per ogni altro prodotto *Adobe*. I campi dato in queste sottodirectory possono contenere qualsiasi informazione che i programmati *Adobe* potrebbero voler inserire, tipicamente, proprietà generiche del sistema come la versione ed il numero di build, come disinstallare il pacchetto, i driver da usare, e così via. Il registro elimina il problema che hanno i programmati di dove immagazzinare tutte queste informazioni; nel registro inoltre risiedono anche alcune informazioni specifiche sull'utente, sotto la chiave HKEY\_USERS.

La sottochiave SYSTEM possiede la maggior parte delle informazioni necessarie all'avvio del sistema, per esempio, la lista dei driver che devono essere caricati. Essa contiene anche la lista dei servizi (demoni) che devono partire dopo l'avvio, e tutte le configurazioni relative.

La chiave successiva di massimo livello è HKEY\_USERS, che contiene i profili di tutti gli utenti. Vengono memorizzate qui tutte le preferenze degli utenti in un certo numero di aree; quando un utente cambia una preferenza utilizzando il pannello di controllo, per esempio il colore dello sfondo del desktop, le nuove regolazioni vengono salvate qui. In effetti, molti dei programmi sul pannello di controllo fanno poco più che raccogliere le informazioni dell'utente e cambiare quelle del registro. Alcune delle sottochiavi di HKEY\_USERS sono mostrate in Figura 11.5 e potrebbero richiedere commenti supplementari; sottochiavi come Software contengono, sorprendentemente, un ampio numero di sottochiavi anche se non vi sono software installati.

La chiave di massimo livello successiva, HKEY\_PERFORMANCE\_DATA, non contiene né dati letti dal disco e neanche dati raccolti dal gestore del plug-and-play, ma offre una finestra sul sistema operativo. Il sistema stesso contiene centinaia di contatori per il monitoraggio delle proprie prestazioni, che sono accessibili attraverso questa chiave di registro. Quando si richiede una sottochiave, si esegue una procedura specifica per raccogliere e restituire le informazioni (eventualmente leggendo uno o più contatori e combinandoli in un certo modo).

Questa chiave non è visibile usando *regedit* o *regedit32*: per vederla occorre usare strumenti di prestazioni come *perfmon*, *perfmon* e *pview*, ma vi sono molti altri strumenti, alcuni nel CD-ROM di Windows 2000, altri nei Pacchetti di Risorse ed anche di altri produttori.

Le successive tre chiavi di massimo livello di fatto non esistono: ognuna è un collegamento simbolico a qualche posto da un'altra parte nel registro. La chiave HKEY\_CLASSES\_ROOT è la più interessante: punta alla directory che gestisce gli oggetti COM (Modello ad oggetti dei componenti - Component Object Model) ed anche le associazioni tra le estensioni dei file ed i programmi. Quando un utente fa doppio click su un file che finisce, ad esempio, con *.doc*, il programma che recuperà il click

| Funzione Win32 API | Descrizione                                                         |
|--------------------|---------------------------------------------------------------------|
| RegCreateKeyEx     | Crea una nuova chiave di registro                                   |
| RegDeleteKey       | Elimina una chiave di registro                                      |
| RegOpenKeyEx       | Apri una chiave per recuperare il gestore                           |
| RegEnumKeyEx       | Enumera le sotto-chiavi subordinate alla chiave relativa al gestore |
| RegQueryValueEx    | Recupera il dato relativo al valore in una chiave                   |

Figura 11.6 Alcune delle chiamate API Win32 per usare il registro.

del mouse controlla qui per capire quale programma lanciare (verosimilmente Microsoft Word). Il database completo delle estensioni riconosciute, e dei programmi associati ad ognuna, si trova sotto questa chiave.

La chiave HKEY\_CURRENT\_CONFIG è collegata alla configurazione hardware corrente; l'utente può costruire differenti configurazioni hardware, per esempio per disabilitare vari dispositivi, se, ad esempio, desidera controllare se sono loro a causare un comportamento anomalo del sistema. Questa chiave punta alla configurazione corrente; ed analogamente, la chiave HKEY\_CURRENT\_USER punta all'utente corrente, in modo da poter trovare facilmente le relative preferenze.

Nessuna delle ultime tre chiavi, in realtà, aggiunge nulla, visto che tutte le informazioni esistevano comunque già altrove nel registro (sebbene accessibili meno agevolmente). Quindi, anche se regedit e regedit32 restituiscono cinque chiavi di livello massimo, di fatto esistono tre chiavi di massimo livello, ed una di queste non è tra le cinque mostrate.

Il registro è disponibile totalmente al programmatore Win32; esistono chiamate per creare o distruggere chiavi, recuperare valori nelle chiavi ed altro ancora; alcune delle chiamate più utili sono listate in Figura 11.6.

Quando il sistema viene spento, la maggior parte delle informazioni di registro (ma non tutte, come già detto in precedenza), viene salvata su disco in file chiamati alveari (hives), per lo più contenuti in \winnt\system32\config. Visto che la loro integrità è critica per il buon funzionamento del sistema, quando vengono aggiornati, si fanno automaticamente anche copie di backup, e le scritture usano transazioni atomiche per impedire che siano corrotte da un crash del sistema nel corso della scrittura. La perdita del registro comporta la reinstallazione di tutto il sistema.

## 11.3 La struttura del sistema

Nelle sezioni precedenti abbiamo esaminato Windows 2000 dal punto di vista di un programmatore; adesso andremo sotto la corteccia per osservare l'organizzazione interna del sistema, cosa realizzano la varie componenti, come interagiscono tra di loro e con i programmi utente. Sebbene molti libri spieghino come usare Windows 2000, ve ne sono molti meno che spiegano come funziona; di gran lunga il miglior posto dove cercare altro materiale è *Inside Windows 2000*, terza edizione, di Solomon e Russinovich (2000). Parte del materiale per questo capitolo è stato preso da tale libro e da informazioni degli autori; anche Microsoft ha fornito molti dati chiave.

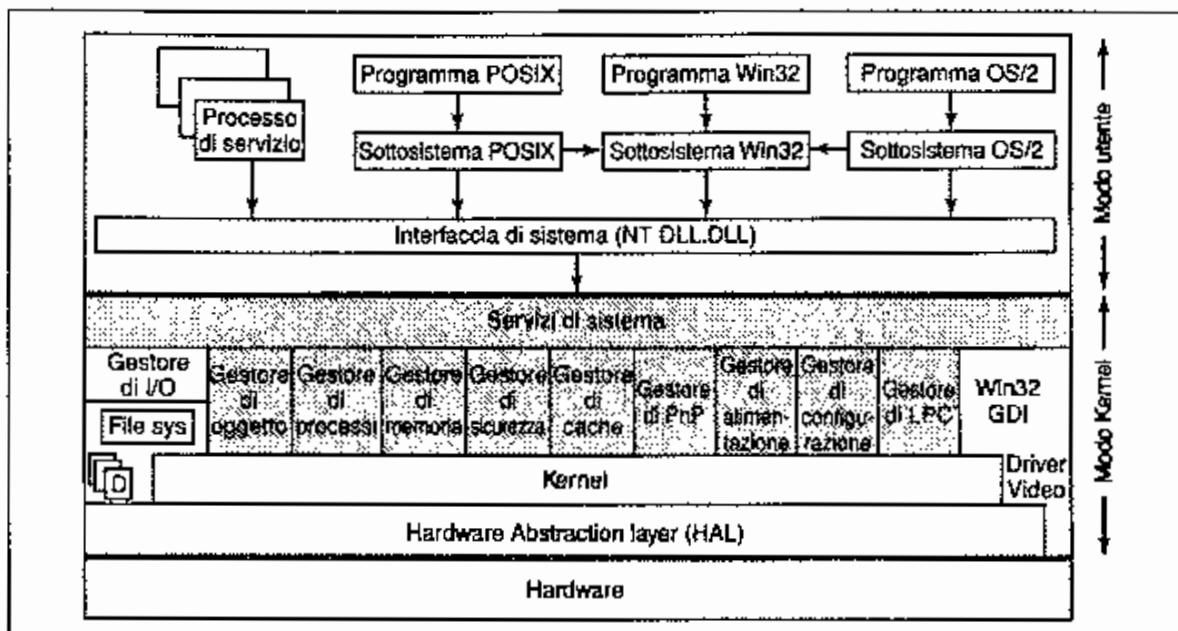


Figura 11.7 La struttura di Windows 2000 (leggermente semplificata). Le aree più scure sono l'esecutivo. Le caselle indicate dalla D sono i driver dei dispositivi. I processi di servizio sono demoni di sistema.

### 11.3.1 La struttura del sistema operativo

Windows 2000 si compone di due pezzi principali: il sistema operativo vero e proprio, che gira in modo kernel, ed i sottosistemi di ambiente, che girano in modo utente. Il kernel è un kernel tradizionale, nel senso che amministra la gestione dei processi, la gestione della memoria, i file system e così via. I sottosistemi di ambiente sono qualcosa di insolito, poiché sono processi separati che aiutano i programmi utente ad attuare le funzioni di sistema. Nelle sezioni seguenti esamineremo a turno ognuna di queste parti.

Uno dei tanti miglioramenti di NT su Windows 3.x è la sua struttura modulare: esso si componeva di un piccolo kernel che girava in modo kernel, più alcuni processi server che giravano in modo utente. I processi utente interagivano con i processi server usando un modello client-server: un client mandava un messaggio ad un server, ed il server faceva il lavoro restituendo il risultato al client con un nuovo messaggio. Questa struttura modulare rendeva più semplice la migrazione del sistema su parecchi computer al di là della linea Intel, come Alpha DEC, IBM PowerPC e SGI MIPS, ed inoltre proteggeva il kernel da problemi nel codice del server. Ad ogni modo, per ragioni di prestazioni, da NT 4.0 in poi, praticamente tutto il sistema operativo (come ad esempio la gestione delle system call e tutta la grafica a video) è stato reinserito nel modo kernel. Il progetto è proseguito in Windows 2000.

Non di meno, esiste ancora un po' di struttura in Windows 2000, che è diviso in parecchi strati, ognuno dei quali utilizza i servizi degli altri al di sotto di lui. La struttura è illustrata in Figura 11.7: ognuno degli strati è diviso orizzontalmente in molti moduli, ciascuno con alcune funzioni peculiari e una interfaccia ben definita con gli altri moduli.

I due strati più in basso nel software, HAL ed il kernel, sono scritti in C e in linguaggio assembler, e sono in parte dipendenti dalla macchina; gli strati superiori sono scritti in C e sono nella quasi totalità indipendenti dalla macchina, i driver sono scritti in C e, in pochi casi, in C++. Di seguito esamineremo le varie componenti del sistema incominciando dagli strati più bassi e salendo man mano.

## Lo strato di astrazione hardware (HAL)

Uno degli scopi di Windows 2000 (e di Windows NT prima di lui) era quello di rendere il sistema operativo portabile tra le varie piattaforme. In linea ideale, quando compare una nuova macchina, dovrebbe bastare ricompilare il sistema operativo con il compilatore della nuova, e mandarlo in esecuzione per la prima volta. Sfortunatamente la vita non è così semplice: mentre gli strati superiori del sistema operativo possono essere completamente portabili (poiché hanno in massima parte a che fare con strutture dati interne), gli strati più in basso hanno a che fare con i registri di dispositivi, le interruzioni, il DMA, ed altre funzionalità di hardware che sono notevolmente diverse da macchina a macchina. Sebbene la maggior parte del codice sia scritto in C, esso non può semplicemente essere preso da un Pentium e spostato, diciamo su Alpha, ricompilato e riavviato, per il fatto che esistono molte piccole differenze di hardware tra il Pentium e Alpha che non hanno nulla a che vedere con i diversi insiemi di istruzioni, e che non possono essere nascoste dal compilatore.

Pienamente consapevole di questo problema, Microsoft fece un tentativo serio per nascondere la maggior parte delle dipendenze dalle macchine in un sottile strato di base chiamato HAL (Hardware Abstraction Layer – Strato di Astrazione Hardware). (Il nome fu senza dubbio preso dal famoso computer HAL del film di Stanley Kubrick dal titolo *2001: Odissea nello Spazio*. Si dice che Kubrick scelse il nome HAL prendendo la sigla IBM – allora l'azienda dominante – e sottraendo 1 da ogni lettera).

Il lavoro di HAL è quello di presentare al resto del sistema operativo dispositivi hardware astratti, in particolare, privi di pecche e idiosincrasie di cui l'hardware è così ampiamente fornito; tali dispositivi sono presentati come servizi indipendenti dalla macchina (chiamate di procedura e macro) che possono essere utilizzati dal resto del sistema operativo e dai driver. Usando i servizi di HAL (che sono identici su tutti i sistemi Windows 2000, non importa quale sia l'hardware) e non indirizzando l'hardware in modo diretto, i driver ed il kernel richiedono solo piccoli cambiamenti per essere portati sul nuovo hardware. Il porting di HAL stesso è immediato, dato che tutto il codice dipendente dalla macchina è concentrato in un posto, e gli obiettivi sono ben definiti, ovvero, implementare nuovamente ogni servizio di HAL.

I servizi forniti da HAL sono collegati all'insieme dei chip delle schede madri, che variano da macchina a macchina all'interno di determinati limiti prevedibili; in altre parole, HAL è progettato per nascondere le differenze fra un produttore di schede madri ed un altro, ma non la differenza tra un'architettura basata su Pentium ed una basata su Alpha. I servizi HAL comprendono l'accesso ai registri dei dispositivi, gli indirizzamenti dei dispositivi indipendenti dal bus, l'inizializzazione e la gestione delle interruzioni, i trasferimenti via DMA, il controllo dei timer e del clock di sistema, la sincronizzazione dei processori e la gestione degli spin lock di basso livello, l'interfaccia con il BIOS e la sua memoria di configurazione CMOS. HAL non fornisce astrazioni o servizi per dispositivi specifici di I/O come tastiere, mouse, dischi o l'unità di gestione della memoria.

Come esempio di ciò che fa lo strato di astrazione hardware, consideriamo il caso di I/O mappato in memoria confrontato con l'uso di porte di I/O: alcune macchine ne hanno un tipo ed altre l'altro. Come si dovrebbe programmare un driver: per usare l'I/O mappato in memoria o no? Piuttosto che forzare una scelta, che renderebbe il driver non portabile su una macchina che opera nel modo diverso, HAL offre ai creatori di driver tre procedure per leggere i registri dei dispositivi, e altre tre per scrivere su questi:

```
uc = READ_PORT_UCHAR(port); WRITE_PORT_UCHAR(port, uc);
us = READ_PORT USHORT(port); WRITE_PORT USHORT(port, us);
ul = READ_PORT ULONG(port); WRITE_PORT ULONG(port, ul);
```

Queste procedure leggono e scrivono rispettivamente interi senza segno di 8-, 16-, 32-bit, nella porta specifica; è compito dello strato di astrazione hardware decidere ogni volta se va utilizzato l'I/O mappato o meno. In questo modo, un driver può essere spostato senza modifiche tra macchine che differiscono nel modo in cui sono implementati i driver dei dispositivi.

I driver hanno spesso bisogno di accedere a dispositivi di I/O specifici per vari scopi: a livello hardware, un dispositivo ha uno o più indirizzi su un certo bus, e poiché i computer moderni hanno spesso più bus (ISA, PCI, SCSI, USB, 1394, eccetera), può succedere che due o più dispositivi abbiano lo stesso indirizzo di bus, e quindi è necessario distinguerli. HAL fornisce un servizio per identificare i dispositivi, mappando gli indirizzi di dispositivo relativi al bus in indirizzi logici nel sistema; in questo modo, i driver non devono più tenere traccia di quale dispositivo si trova sul bus. Questi indirizzi logici sono analoghi ai gestori che il sistema fornisce ai programmi utenti per riferirsi a file e ad altre risorse di sistema, e tale meccanismo protegge anche gli strati più alti dalle proprietà di strutture di bus alternative e da convenzioni di indirizzamento.

Le interruzioni hanno un problema simile – sono anche esse dipendenti dal bus; ed anche qui HAL fornisce servizi per nominare le interruzioni in modo generico, oltre a servizi per permettere ai driver di collegare le routine di servizio di interruzioni in modo portabile, senza dover conoscere nulla sul vettore di interruzioni associato ad un certo bus. Anche la gestione del livello di richiesta di interruzioni viene amministrata da HAL.

Un altro servizio offerto da HAL è la definizione e la gestione dei trasferimenti DMA in modo indipendente dai dispositivi; possono essere amministrati sia il DMA di sistema sia i DMA su specifiche schede di I/O. I dispositivi sono indirizzati mediante indirizzi logici; HAL implementa anche via software la dispersione ed il raggruppamento (scrivendo o leggendo da blocchi non contigui di memoria fisica).

Inoltre, HAL gestisce i clock ed i timer in modo portabile; il tempo viene mantenuto in unità di 100 ns ad incominciare dal 1 Gennaio 1601, che è molto più preciso di quello che faceva MS-DOS partendo dal 1 Gennaio 1980 ed utilizzando unità di 2 secondi, e viene fornito supporto per molte attività legate ai computer nei secoli 17-esimo, 18-esimo e 19-esimo. I servizi di tempo sganciano i driver dalle frequenze effettive a cui viaggiano i clock.

Le componenti del kernel talvolta hanno necessità di sincronizzarsi a basso livello, in special modo per impedire corse critiche nei sistemi multiprocessore. HAL fornisce alcune primitive per gestire questa sincronizzazione, come gli spin lock, nei quali una CPU attende semplicemente che sia rilasciata una risorsa tenuta da un'altra CPU, usati tipicamente in casi in cui la risorsa è tenuta solo per poche istruzioni macchina.

Infine, dopo l'avvio del sistema, HAL parla con il BIOS ed esamina la configurazione di memoria CMOS, se presente, per scoprire quali bus e dispositivi di I/O sono presenti nel sistema, e come sono stati configurati. Queste informazioni sono quindi messe nel registro, in modo che le altre componenti del sistema possano recuperarle senza capire come lavorino il BIOS e la configurazione della memoria. Un riassunto dei compiti di HAL si trova in Figura 11.8.

Poiché HAL è fortemente dipendente dalla macchina, deve combaciare perfettamente con il sistema in cui è installato e, per questo, Windows 2000 è provvisto nel CD-ROM di parecchi HAL. Al momento dell'installazione del sistema, l'HAL appropriato viene scelto e copiato nella directory \winnt\system32 sull'hard disk come hal.dll. Tutti gli avvii successivi usano questa versione di HAL; rimuovere questo file non permetterebbe più al sistema di avviarsi.

Sebbene HAL sia abbastanza efficiente, potrebbe non essere abbastanza veloce per le applicazioni multimediali, per questa ragione Microsoft ha prodotto un pacchetto

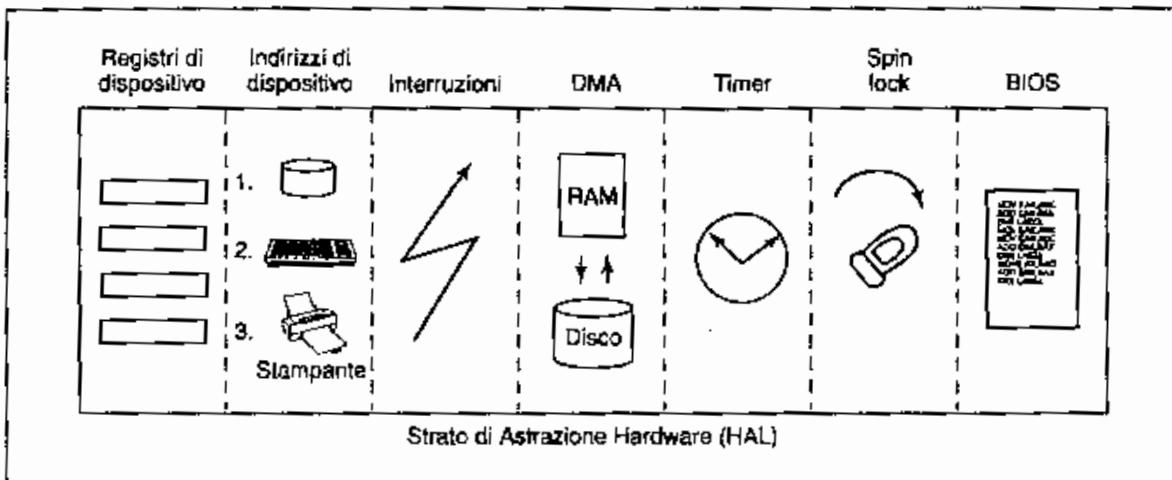


Figura 11.8 Alcune delle funzioni hardware gestite da HAL.

software, DirectX, che aumenta le prestazioni di HAL con altre procedure e permette ai processi utente di avere un accesso più diretto all'hardware. DirectX è qualcosa di specialistico e per questo verrà trattato in questo capitolo.

## Lo strato kernel

Al di sopra dello strato di astrazione hardware vi è uno strato che contiene quello che Microsoft chiama il **kernel**, oltre ai driver di dispositivi. Alcune documentazioni precedenti facevano riferimento al kernel come al "micro-kernel", ma non era affatto "micro" perché il gestore della memoria, il file system, ed altre componenti principali risiedono nello spazio kernel, e girano in modo kernel da sempre. Il kernel non è di certo un micro-kernel adesso, poiché virtualmente l'interno sistema operativo era inserito nello spazio kernel già da NT 4.0.

Nel capitolo di UNIX, avevamo utilizzato il termine "kernel" per indicare ogni cosa che girasse in modo kernel; in questo capitolo (molto a malincuore) riserveremo questo termine per la parte etichettata così in Figura 11.7 e chiameremo la totalità del codice che gira in modo kernel come "sistema operativo". Parte del kernel (e la maggior parte di HAL) è sempre residente in memoria (cioè non è paginata); aggiustando la propria priorità, è in grado di controllare se e quando può essere prerilasciata da interruzioni di I/O. Sebbene una parte consistente del kernel sia specifica della macchina, la maggior parte è scritta in C, eccetto nei casi in cui la prestazione massima oscura tutte le altre problematiche.

Lo scopo del kernel è quello di rendere tutto il resto del sistema operativo indipendente dall'hardware e quindi portabile; esso incomincia dove HAL smette, accede all'hardware attraverso HAL, e costruisce astrazioni ad alto livello sui servizi di più basso livello di HAL. Per esempio, HAL possiede chiamate per associare procedure di servizi di interruzione con interruzioni, e per definire le priorità, ma fa ben poco altro in quest'area. Il kernel, per contro, fornisce un meccanismo completo per fare il cambio di contesto (context switch): salva in modo corretto tutti i registri di CPU, cambia le tabelle delle pagine, svuota la cache della CPU, e così via e, quando ha fatto questo, il thread che era in esecuzione in precedenza è stato salvato in tabelle in memoria. Dopodiché, definisce la nuova mappa di memoria del nuovo thread e carica i suoi registri, affinché il nuovo thread possa andare in esecuzione.

Il codice per lo scheduling dei thread è anch'esso nel kernel: quando è il momento di vedere se un nuovo thread può andare in esecuzione, nel caso ad esempio in cui è scaduto un quanto, o dopo una interruzione di I/O, il kernel sceglie il thread, e fa il cambio di contesto necessario perché vada in esecuzione. Dal punto di vista del resto del sistema operativo, la commutazione di thread viene gestita in automatico dagli strati più bassi, e in modo portabile, senza altri interventi. L'algoritmo di scheduling verrà trattato più avanti in questo capitolo quando arriveremo ai thread ed ai processi.

Oltre a fornire una astrazione ad alto livello dell'hardware, e gestire la commutazione di thread, il kernel possiede un'altra funzione chiave: fornisce un supporto a basso livello per due classi di oggetti: gli oggetti di controllo e gli oggetti di smistamento. Questi oggetti non sono oggetti sui quali i processi utente possono ottenere un gestore, ma sono oggetti sui quali l'esecutivo costruisce gli oggetti utente.

Gli oggetti di controllo (**Control Object**) sono quegli oggetti che controllano il sistema, tra cui gli oggetti di processo primitivo, gli oggetti di interruzione, e due altri strani oggetti chiamati DPC e APC. Un oggetto DPC (**Chiamata di Procedura Rinviabile - Deferred Procedure Call**) viene usato per dividere la parte che non è critica in tempo di una procedura di servizio di interruzioni, dalla parte che invece lo è. In genere, una procedura di servizio di interruzioni salva pochi registri hardware volatili associati al dispositivo di I/O che fa l'interruzione, perché non siano sovrascritti, e riabilita l'hardware, ma rimanda a dopo il grosso dell'elaborazione.

Per esempio, dopo che è stato premuto un tasto, la procedura di servizio di interruzioni della tastiera legge il codice del tasto da un registro e riabilita l'interruzione da tastiera, ma non ha bisogno di elaborare il tasto immediatamente, specialmente se sta facendo qualcosa di più importante (cioè con priorità più alta): se il tasto verrà elaborato entro circa 100 ms, l'utente non si accorgerà mai del ritardo. I DPC sono anche usati nel caso di scadenze dei timer, e di altre attività per le quali l'elaborazione non deve essere necessariamente istantanea; la coda DPC è un meccanismo per ricordarsi che vi è del lavoro da fare più tardi.

Un altro oggetto di controllo del kernel è APC (**Chiamata di Procedura Asincrona - Asynchronous Procedure Call**). Gli oggetti APC sono simili agli oggetti DPC, però vengono eseguiti nel contesto di un processo specifico: quando si deve gestire la pressione di un tasto, non importa in quale contesto si esegue il DPC, perché quello che ci si aspetta è che il codice del tasto sia esaminato e posto probabilmente nel buffer del kernel. Se invece una interruzione richiede di copiare il buffer del kernel in un buffer nello spazio degli indirizzi di un processo (ad esempio, in fase di completamento di lettura da un modem), la procedura di copiatura ha bisogno di girare nel contesto del ricevente, quindi la tabella delle pagine dovrà contenere tutti e due i buffer, kernel e utente (tutti i processi contengono tutto il kernel nel loro spazio di indirizzi, come vedremo più avanti). Per questa ragione, il kernel distingue tra DPC e APC.

Gli altri tipi di oggetti kernel sono gli oggetti di smistamento (**dispatcher object**), tra cui vi sono i semafori, i mutex, gli eventi, i timer di attesa, ed altri oggetti per i quali i thread possono essere posti in stato di attesa. Il motivo per cui questi oggetti devono essere amministrati (in parte) dal kernel è che essi sono intimamente collegati con lo scheduling dei thread, il che è un compito assegnato al kernel. Tra parentesi, i mutex sono chiamati "mutanti" nel codice perché è stato loro imposto di implementare la semantica di OS/2, cioè di non sbloccarsi automaticamente quando un thread che ne detiene uno sta aspettando, cosa che i progettisti di Windows 2000 hanno considerato bizzarra. (La semantica di OS/2 è importante poiché NT fu originariamente concepito come sostituto di OS/2, il sistema operativo venduto sul PC/2 della IBM).

## L'esecutivo

Sopra il kernel ed i driver di dispositivi si trova la porzione più alta di sistema operativo chiamata **esecutivo (executive)**, mostrato come una casella ombreggiata in Figura 11.7. L'esecutivo è scritto in C, è indipendente dall'architettura, e può essere portato su di una nuova macchina con uno sforzo abbastanza piccolo. Esso è fatto da 10 componenti, ognuna della quali è semplicemente una collezione di procedure che lavorano insieme per raggiungere uno scopo comune. Non esiste un confine preciso tra i vari pezzi, e diversi autori potrebbero perfino raggruppare le procedure in modo diverso. Occorre notare che componenti che si trovano allo stesso livello possono chiamarsi a vicenda illimitatamente (e lo fanno).

Il **gestore di oggetti (object manager)** gestisce tutti gli oggetti che sono conosciuti dal sistema operativo, come i processi, i thread, i file, le directory, i semafori, i dispositivi di I/O, i timer e molti altri. Quando viene creato un oggetto, il gestore di oggetti alloca un blocco di memoria virtuale dallo spazio degli indirizzi del kernel, poi, se viene deallocated, lo restituisce alla lista libera; il suo compito è di tenere traccia di tutti gli oggetti.

Per evitare qualsiasi confusione, la maggior parte delle componenti dell'esecutivo etichettate con "gestore" in Figura 11.7 non sono processi o thread, ma semplicemente collezioni di procedure che altri thread possono eseguire quando sono in modo kernel. Alcune però sono thread completamente indipendenti, come il gestore dell'alimentazione ed il gestore del plug-and-play.

Il gestore degli oggetti gestisce anche uno spazio dei nomi, nel quale dispone gli oggetti creati in modo che a questi ci si possa riferire in un secondo momento. Tutte le altre componenti dell'esecutivo utilizzano fortemente gli oggetti per svolgere il proprio lavoro; gli oggetti sono così centrali nel funzionamento di Windows 2000 che verranno trattati in dettaglio più avanti nella prossima sezione.

Il **gestore di I/O (I/O manager)** fornisce un ambiente per la gestione dei dispositivi di I/O ed altri servizi generici di tipo I/O, dando al resto del sistema un I/O indipendente dal dispositivo, chiamando il driver appropriato ad effettuare l'I/O fisico; è inoltre l'ospite di tutti i driver di dispositivi (indicati da D in Figura 11.7). I file system sono tecnicamente dei driver di dispositivi sotto il controllo del gestore di I/O; ve ne sono due diversi, per il file system FAT e NTFS, ognuno indipendente dall'altro per controllare partizioni differenti, e tutti i file system FAT sono gestiti da un singolo driver. Studieremo l'I/O più avanti nella sezione 11.6 ed uno dei file system, NTFS, nella sezione 11.7.

Il **gestore dei processi (process manager)** amministra i processi ed i thread, compresa la loro creazione e la loro terminazione; contiene i meccanismi utilizzati per gestirli, piuttosto che le politiche su come questi devono essere usati. Esso si basa sul processo kernel e sugli oggetti thread e vi aggiunge altre funzionalità, ed è la chiave per la multiprogrammazione in Windows 2000. Studieremo la gestione dei processi e dei thread nella sezione 11.4.

Il **gestore della memoria (memory manager)** implementa l'architettura di memoria virtuale paginata a richiesta di Windows 2000; amministra la mappa tra pagine virtuali e pagine fisiche; controlla, quindi, le regole di protezione che costringono ogni processo ad accedere solo alle pagine che appartengono al suo spazio di indirizzi e non agli spazi di indirizzi di altri processi (tranne che in certe circostanze), ed amministra anche certe chiamate di sistema che sono legate alla memoria virtuale. Studieremo la gestione della memoria in sezione 11.5.

Il **gestore della sicurezza (security manager)** realizza l'elaborato meccanismo di sicurezza di Windows 2000, che rispetta i requisiti C2 dell'Orange Book del Dipartimento della Difesa Americano. L'Orange Book specifica un gran numero di regole che un sistema conforme deve rispettare, a partire dall'accesso al sistema mediante autenticazione fino

ad arrivare a come amministrare il controllo degli accessi ed al fatto che le pagine virtuali devono essere azzerate prima di essere riutilizzate. Studieremo la gestione della sicurezza in Windows 2000 nella sezione 11.8.

Il gestore di cache (**cache manager**) mantiene in memoria i blocchi disco usati recentemente, per accelerare l'accesso agli stessi nel caso (probabile) in cui siano ancora necessari. Il suo lavoro è quello di scoprire quali blocchi saranno necessari al passo successivo e quali no; è possibile configurare Windows 2000 con file system multipli, nel qual caso il gestore di cache lavora per tutti loro in modo che non debbano preoccuparsi della propria gestione di memoria. Quando un blocco è necessario, esso viene chiesto al gestore di cache; e se non possiede il blocco, lo deve richiedere al file system appropriato. Poiché i file possono essere mappati negli spazi degli indirizzi dei processi, il gestore della cache deve poter interagire con il gestore della memoria virtuale per garantire la consistenza. La quantità di spazio dedicata alle operazioni di caching è dinamica, e può essere aumentata e diminuita se variano le richieste alla stessa; studieremo la gestione della cache in sezione 11.9.

Il gestore del plug-and-play riceve notifiche per ogni nuovo dispositivo che viene connesso; per alcuni dispositivi, si fa un controllo solo al momento dell'avvio e in nessun altro momento, ma altri dispositivi, per esempio i dispositivi USB, possono essere attaccati in qualsiasi momento, ed il loro collegamento causa immediatamente un messaggio al gestore del plug-and-play, che li individua e carica i driver appropriati.

Il gestore di alimentazione (**power manager**) gestisce l'uso dell'alimentazione, come lo spegnere lo schermo ed il disco dopo un certo tempo di inutilizzo. Sui computer portatili, il gestore di alimentazione controlla l'uso della batteria ed agisce quando la batteria sta per finire la carica. Solitamente spedisce un messaggio ai programmi, per salvare i propri file, e per prepararsi ad un probabile spegnimento immediato del sistema.

Il gestore di configurazione (**configuration manager**) ha in carico il registro; vi aggiunge nuovi campi dato e recupera le chiavi quando richiesto.

Il gestore di chiamate di procedure locali (**local procedure call manager**) fornisce una comunicazione tra processi altamente efficiente, usata dai processi stessi e dai loro sottosistemi. Poiché il suo servizio è fondamentale per attuare alcune chiamate di sistema, l'efficienza è un fattore molto importante, e per questo non utilizza i meccanismi tradizionali di comunicazione tra processi.

Il modulo esecutivo GDI Win32 amministra certe chiamate di sistema (ma non tutte); si trovava originariamente nello spazio utente ma è stato spostato nello spazio kernel (da NT 4.0) per migliorarne le prestazioni. Il GDI (Graphics Device Interface – Interfaccia ai dispositivi grafici) gestisce le immagini per lo schermo e le stampanti; fornisce delle chiamate di sistema per permettere ai programmi utente di scrivere sullo schermo e sulle stampanti in modo indipendente dal dispositivo, e contiene anche il gestore delle finestre ed i driver dello schermo. Prima di NT 4.0 esso si trovava nello spazio utente, ma le prestazioni erano deprimenti, così Microsoft lo ha spostato nello spazio kernel per velocizzarlo. È importante sottolineare che la Figura 11.7 non è affatto in scala: per esempio, i moduli di interfaccia Win32 verso i dispositivi grafici sono più grandi di tutto il resto dell'esecutivo.

In cima all'esecutivo si trova un sottile strato chiamato servizi di sistema (**system services**). La sua funzione è fornire un'interfaccia all'esecutivo; accetta le voci chiamate di sistema di Windows 2000 e chiama le altre parti dell'esecutivo perché le eseguano.

Al momento dell'avvio del sistema, Windows 2000 viene caricato nella memoria come una collezione di file; la parte fondamentale del sistema operativo, comprendente il kernel e l'esecutivo, si trova nel file *ntoskrnl.exe*. HAL è una libreria condivisa posta in *hal.dll*; l'interfaccia ai dispositivi grafici e Win32 si trovano insieme in un terzo file, *win32k.sys*. Infine vengono caricati anche molti altri driver di dispositivi, per lo più con estensione *.sys*.

In realtà, le cose non sono così semplici: il file *ntoskrnl.exe* arriva in versione uniprocesso e multiprocessore, e vi sono versioni per Xeon, che ha più di 4GB di memoria fisica e per Pentium, che non può averla; inoltre, le versioni possono essere free build (vendute nei negozi ed installate dai produttori) o build controllate (per scopi di debugging). In tutto vi possono essere otto combinazioni, ma due sono state inglobate in altre e per questo ne rimangono solo sei; una di queste viene copiata nell'*ntoskrnl.exe* quando il sistema viene installato.

Le build controllate meritano un paio di parole in più: quando un nuovo dispositivo di I/O viene installato in un PC, vi è innanzitutto un driver fornito dal produttore, che deve essere installato perché il dispositivo funzioni. Supponiamo che la scheda IEEE 1394 venga installata su un computer e senz'altro funzionare perfettamente, ma due settimane più tardi il sistema all'improvviso di arresta. Chi incolperà l'utente? La Microsoft.

Il problema può essere certamente dovuto a Microsoft, ma alcuni problemi possono dipendere da driver mal funzionanti, sui quali Microsoft non ha controllo, che sono stati installati in memoria kernel ed hanno completo accesso a tutte le tabelle kernel come all'intero hardware. Tentando di diminuire il numero di clienti adirati al telefono, Microsoft cerca di aiutare i programmati di driver nella messa a punto del loro codice mettendo espressioni della forma

#### **ASSERT (condizione)**

nel codice. Queste espressioni fanno controlli di correttezza su tutti i parametri delle procedure interne al kernel (che possono essere liberamente chiamate dai driver) come anche molti altri controlli. Le free build possiedono *ASSERT* definita come macro che non fa nulla, rimuovendo ogni controllo; le build controllate l'hanno definita come:

```
#define ASSERT(a) if (!(a)) error(...)
```

provocando la presenza di tutti i controlli nel codice eseguibile di *ntoskrnl.exe* e la loro esecuzione. Tutto questo rallenta enormemente il sistema, ma aiuta i programmati di driver a controllarli prima di venderli ai clienti. Le build controllate hanno anche numerose altre funzionalità di debug.

## I driver di dispositivi

L'ultima parte di Figura 11.7 contiene i **driver di dispositivi** (device drivers). Ogni driver di dispositivo può controllare uno o più dispositivi di I/O, ma un driver di dispositivo può anche fare cose non legate ad uno specifico dispositivo, come cifrare un flusso di dati o anche fornire accesso alle strutture dati del kernel. I driver di dispositivi non sono parte del binario *ntoskrnl.exe*, il vantaggio di questa scelta è che una volta che i driver sono stati installati nel sistema, li si aggiunge nella lista del registro, e li si carica dinamicamente quando il sistema viene avviato. In questo modo *ntoskrnl.exe* è lo stesso per tutti, ma ogni sistema viene configurato in modo preciso rispetto ai dispositivi di cui è fornito.

Vi sono driver di dispositivi per dispositivi macroscopici, come nel caso di dischi e stampanti, ma anche per dispositivi interni e circuiti integrati dei quali nessuno ha mai sentito parlare; in più i file system sono presenti come se fossero driver di dispositivi, come già detto. Il più grande driver di dispositivo, quello per Win32, GDI, e per il video, è mostrato all'estrema destra di Figura 11.7; esso gestisce molte chiamate di sistema e la maggior parte della grafica. Poiché i clienti possono installare nuovi driver di dispositi-

vi, essi hanno anche il potere di influenzare il kernel e contaminare il sistema; per questa ragione i driver devono essere scritti con molta attenzione.

### 11.3.2 Implementazione degli oggetti

Gli oggetti sono probabilmente il concetto più importante di Windows 2000; essi forniscono un'interfaccia uniforme e consistente a tutte le risorse di sistema ed alle strutture dati come i processi, i thread, i semafori, eccetera. Questa uniformità ha varie sfacciate: in primo luogo, tutti gli oggetti vengono nominati e acceduti allo stesso modo, usando i gestori di oggetti; in secondo luogo, poiché tutti gli accessi ad un oggetto avvengono tramite il gestore dell'oggetto, è possibile mettere tutti i controlli di sicurezza in un solo punto per assicurarsi che nessun processo possa evitarli. Terzo, la condivisione di oggetti tra i processi può essere gestita in modo uniforme. Quarto, poiché ogni apertura e chiusura dell'oggetto avviene attraverso il gestore degli oggetti, è facile tenere traccia di quali oggetti sono ancora in uso e quali possono essere cancellati in modo sicuro. Quinto, questo modello uniforme per la gestione degli oggetti rende facile modellare le quote di risorse in modo praticamente immediato.

Una chiave per comprendere gli oggetti è pensare che un oggetto (nell'esecutivo) è solo un certo numero di parole consecutive in memoria (cioè nello spazio degli indirizzi virtuali del kernel); un oggetto è una struttura dati in RAM, nulla di più e nulla di meno. Un file sul disco non è un oggetto, sebbene si crei un oggetto (ovvero una struttura dati nello spazio degli indirizzi virtuali del kernel) per il file quando questo viene aperto. Una conseguenza del fatto che gli oggetti sono semplicemente strutture dati del kernel è che quando il sistema viene riavviato (o si arresta) tutti gli oggetti vengono perduti. In effetti, quando il sistema si avvia non vi è alcun tipo di oggetto (fatta eccezione per i processi di sistema ed inattivi, per i quali gli oggetti sono modellati nel file *ntoskrnl.exe*); tutti gli altri oggetti vengono creati in linea non appena il sistema viene avviato, e partono i vari programmi di inizializzazione (e poi quelli degli utenti).

Gli oggetti hanno una struttura mostrata in Figura 11.9; ogni oggetto contiene una intestazione (header) con informazioni comuni a tutti gli oggetti di quel tipo. I campi di

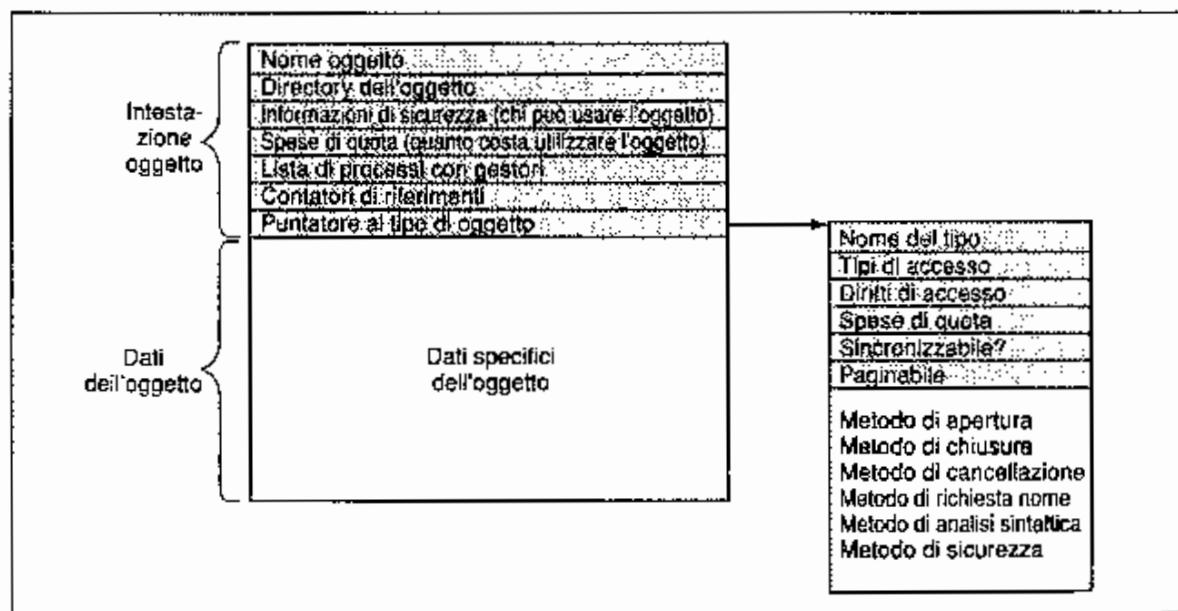


Figura 11.9 La struttura di un oggetto.

questa intestazione comprendono il nome dell'oggetto, la directory in cui vive nello spazio degli oggetti, informazioni di sicurezza (in modo da poter fare un controllo quando un oggetto viene aperto), e una lista di processi con gestori aperti per l'oggetto (se un certo flag di debugging è attivo).

Ogni intestazione di oggetto contiene anche un campo di carico di quota, che è un carico imposto al processo per aprire l'oggetto: se un oggetto file costa 1 punto, ed un processo appartiene ad un job che ha 10 punti di file come valore di quota, i processi in quel job possono aprire solo 10 file in totale. In questo modo si può ottenere un controllo separato sui limiti delle risorse utilizzate per ogni tipo di oggetto.

Gli oggetti occupano un certo spazio - parti di spazio degli indirizzi virtuale del kernel - e quindi, quando un oggetto non è più necessario deve essere prontamente rimosso per ricuperare il suo spazio degli indirizzi. Il meccanismo di recupero prevede un contatore di riferimenti in ogni intestazione di oggetto, che conta il numero di gestori aperti tenuti dai processi. Questo contatore viene incrementato ogni volta che l'oggetto viene aperto, e decrementato ogni volta che viene chiuso; quando il suo valore è 0, non esistono più utenti che desiderano manipolare l'oggetto. Quando un oggetto viene acquisito o rilasciato da una componente dell'esecutivo, un secondo contatore viene incrementato o decrementato, anche quando non vi sono di fatto gestori dell'oggetto; quando entrambi i contatori sono a 0, non vi sono né processi utente né processi esecutivi che stanno utilizzando l'oggetto e quindi questo può essere rimosso per liberare la memoria.

Il gestore degli oggetti ha bisogno di mantenere delle strutture dati dinamiche (i suoi oggetti), ma non è la sola parte dell'esecutivo con queste necessità: anche altre parti hanno bisogno di allocare e liberare dinamicamente parti di memoria di kernel. Per soddisfare queste necessità, l'esecutivo mantiene due pool di pagine nello spazio degli indirizzi del kernel: per gli oggetti e per le altre strutture dati dinamiche. Questi pool operano come heap, cioè come le chiamate *malloc* e *free* del linguaggio C per la gestione dei dati dinamici; un pool viene paginato e l'altro viene bloccato in memoria. Gli oggetti dei quali si ha bisogno spesso sono messi nel pool non-paginato; gli oggetti a cui si accede raramente, come le chiavi di registro ed altre informazioni di sicurezza, sono presenti nel pool paginato. Quando la memoria è poca, questi ultimi possono essere paginati fuori e richiamati da un fault a richiesta; in effetti, grandi porzioni di codice e di strutture dati del sistema operativo sono paginabili, per ridurre il consumo di memoria. Gli oggetti che possono servire quando il sistema sta facendo girare del codice critico (e quando la paginazione non è permessa) devono andare nel pool non paginato. Quando è necessaria poca memoria, una pagina può essere presa da un pool e poi divisa in unità più piccole di 8 byte.

Gli oggetti sono tipati, che vuol dire che ognuno ha proprietà comuni a tutti gli oggetti del suo stesso tipo; il tipo è indicato da un puntatore nell'intestazione ad un tipo di oggetto come mostrato in Figura 11.9. Le informazioni sul tipo di oggetto comprendono: il nome del tipo, se un thread può aspettare l'oggetto (Sì per mutex, NO per file aperti), e se un nuovo oggetto di questo tipo deve andare nel pool paginato o meno. Ogni oggetto punta al suo tipo di oggetto.

L'ultima cosa che hanno i tipi di oggetti è anche la più importante: i puntatori ai codice per certe operazioni standard come *open*, *close* e *delete*. Ogni volta che si chiama una di queste operazioni per un oggetto, si segue il puntatore al tipo di oggetto per trovare il codice relativo ed eseguirlo. Questo meccanismo fornisce al sistema la possibilità di creare nuovi oggetti e recuperare memoria quando questi vengono cancellati.

Le componenti dell'esecutivo possono creare nuovi tipi dinamicamente; non esiste una lista definitiva di tipi di oggetti, ma alcuni tra i più comuni sono elencati in Figura 11.10: vediamoli un po' in dettaglio. I processi ed i thread sono ovvi: vi è un oggetto per ogni processo ed uno per ogni thread, che contiene le proprietà principali necessarie

| Tipo                   | Descrizione                                                                         |
|------------------------|-------------------------------------------------------------------------------------|
| Processo               | Processo utente                                                                     |
| Thread                 | Thread in un processo                                                               |
| Semaforo               | Semaforo a contatore utilizzato per la sincronizzazione tra processi                |
| Mutex                  | Semaforo binario utilizzato per entrare in una regione critica                      |
| Evento                 | Oggetto di sincronizzazione con stato persistente (segnalato/non segnalato)         |
| Porta                  | Meccanismo per lo scambio di messaggi tra processi                                  |
| Timer                  | Oggetto che permette ad un thread di dormire per un intervallo di tempo fissato     |
| Coda                   | Oggetto usato per la notifica di completamento in caso di I/O sincrono              |
| File aperto            | Oggetto associato con un file aperto                                                |
| Token di accesso       | Descrittore di sicurezza per oggetto                                                |
| Profilo                | Struttura dati utilizzata per fare il profilo dell'uso della CPU                    |
| Sezione                | Struttura utilizzata per le mappature di file nello spazio degli indirizzi virtuali |
| Chiave                 | Chiave di registro                                                                  |
| Directory dell'oggetto | Directory per raggruppare gli oggetti nel gestore di oggetti                        |
| Collegamento simbolico | Puntatore ad un altro oggetto per nome                                              |
| Dispositivo            | Oggetto di dispositivo di I/O                                                       |
| Driver di dispositivo  | Ogni driver di dispositivo caricato ha il proprio oggetto associato                 |

Figura 11.10 Alcuni tipi comuni degli oggetti dell'esecutivo gestiti dal gestore degli oggetti.

per gestire il processo o il thread. I tre oggetti successivi, semafori, mutex ed eventi, hanno tutti a che fare con la sincronizzazione tra processi: i semafori ed i mutex lavorano come ci si aspetta, ma con vari campanelli e fischietti extra (ad esempio, valore massimo e timeout); gli eventi possono essere in uno di due possibili stati: trasmessi o non trasmessi. Se un thread aspetta un evento che si trova in stato trasmesso, il thread viene rilasciato immediatamente; se l'evento si trova in stato non trasmesso, il thread si blocca fino a quando un qualche altro thread non trasmette l'evento, il che rilascia tutti i thread bloccati. Un evento può anche cambiare di stato, cioè dopo aver aspettato con successo un segnale, passa automaticamente allo stato non trasmesso, e non rimane nello stato trasmesso.

Gli oggetti porta, timer e coda sono anche loro collegati alla comunicazione e sincronizzazione; le porte sono canali tra processi per lo scambio di messaggi; i timer forniscono un modo per bloccare un processo per un intervallo di tempo specifico; le code sono utilizzate per notificare ai thread che è stata completata un'operazione di I/O, iniziata precedentemente in modo asincrono.

Gli oggetti di file aperti sono creati quando un file viene aperto; i file non aperti non hanno alcun oggetto manipolabile dal gestore di oggetti. I token di accesso sono oggetti legati alla sicurezza; essi identificano un utente e dicono quali privilegi esso ha, se ne ha. I profili sono strutture usate per memorizzare campioni periodici del contatore di programma del thread in esecuzione, in modo da vedere dove il programma sta passando il suo tempo.

Le sezioni sono oggetti usati dal gestore delle memoria per manipolare i file mappati in memoria; esse registrano la mappa tra file (o parti di questo) e gli indirizzi di

memoria. Le chiavi sono chiavi di registro e sono utilizzate per mettere in relazione nomi e valori. Le directory degli oggetti sono interamente locali al gestore degli oggetti; esse forniscono, esattamente nello stesso modo in cui le directory operano nel file system, un meccanismo per raggruppare oggetti correlati. Anche i collegamenti (link) simbolici sono simili alle loro controparti nel file system: essi permettono ad un nome in una certa parte dello spazio dei nomi degli oggetti di riferirsi ad un oggetto in una parte diversa dello spazio dei nomi. Ogni dispositivo conosciuto ha un oggetto dispositivo che contiene le informazioni relative, ed è usato per riferirsi al dispositivo all'interno del sistema. Infine, ogni driver di dispositivo che è stato caricato ha un proprio oggetto nello spazio degli oggetti.

Gli utenti possono creare nuovi oggetti o aprire esistenti facendo chiamate Win32 come `CreateSemaphore` o `OpenSemaphore`; queste sono chiamate a procedure di libreria che in definitiva provocano chiamate di sistema appropriate. Il risultato di una qualsiasi chiamata che crea o apre un oggetto con successo è un campo dato di 64 bit nella tabella dei gestori, che viene memorizzato nella tabella dei gestori privata del processo, nella memoria del kernel. Per usarlo nelle chiamate successive, viene restituito all'utente l'indice (di 32-bit) nella tabella dei gestori.

Il campo dato di 64-bit nella tabella dei gestori contiene due parole da 32-bit; una contiene un puntatore (di 29-bit) all'intestazione dell'oggetto, mentre i tre bit di ordine basso sono usati come flag (ad esempio, se il gestore può venire ereditato da un processo figlio); questi bit sono cancellati prima di usare il puntatore. L'altra parola contiene una maschera dei diritti a 32-bit, necessaria poiché i controlli di permessi sono fatti solo al momento in cui l'oggetto viene creato o aperto. Se un processo ha solo i permessi in lettura di un oggetto, tutti gli altri bit della maschera dei diritti saranno messi a 0, dando al sistema operativo la possibilità di rifiutare una qualsiasi richiesta diversa dalla lettura.

Le tabelle dei gestori per due processi e le loro relazioni con alcuni oggetti sono mostrate in Figura 11.11. In questo esempio, il processo A ha accesso ai thread 1 e 2 ed ai mutex 1 e 2; il processo B ha accesso al thread 3 ed ai mutex 2 e 3. I campi dato corrispondenti nelle tabelle dei gestori contengono i diritti per ognuno di questi oggetti: per esempio, il processo A potrebbe avere i diritti di bloccare e sbloccare i suoi mutex, ma non il permesso di distruggerli. Si noti che il mutex 2 è condiviso da entrambi i processi, permettendo ai thread di sincronizzarsi; gli altri mutex non sono condivisi, il che potrebbe significare che i thread nel processo A usano il mutex 1 per la loro sincronizzazione interna, ed i thread in B utilizzano il mutex 3 per la loro.

## Lo spazio dei nomi di oggetti

Poiché gli oggetti sono creati e distrutti in fase di esecuzione, il gestore degli oggetti deve avere un modo per tenerne traccia: per fare questo, esso mantiene uno spazio dei nomi, nel quale inserisce tutti gli oggetti nel sistema. Lo spazio dei nomi può essere usato da un processo per localizzare e aprire un gestore per gli oggetti di qualche altro processo, ammesso che ne abbia il diritto. Lo spazio dei nomi di oggetti è uno dei tre spazi di nomi mantenuti da Windows 2000; gli altri sono gli spazi di nomi del file system e quello del registro. Tutti e tre sono gerarchici, con livelli multipli di directory per organizzare i campi dato. Gli oggetti directory elencati in Figura 11.10 forniscono il modo per implementare questo spazio di nomi gerarchico per gli oggetti.

Poiché gli oggetti esecutivi sono volatili (ovvero svaniscono quando il computer viene spento, diversamente dai campi dato del file system e del registro), quando il sistema si avvia non vi sono oggetti in memoria e lo spazio dei nomi degli oggetti è vuoto. Durante la fase di avvio, varie parti dell'esecutivo creano directory e le riempiono con oggetti;

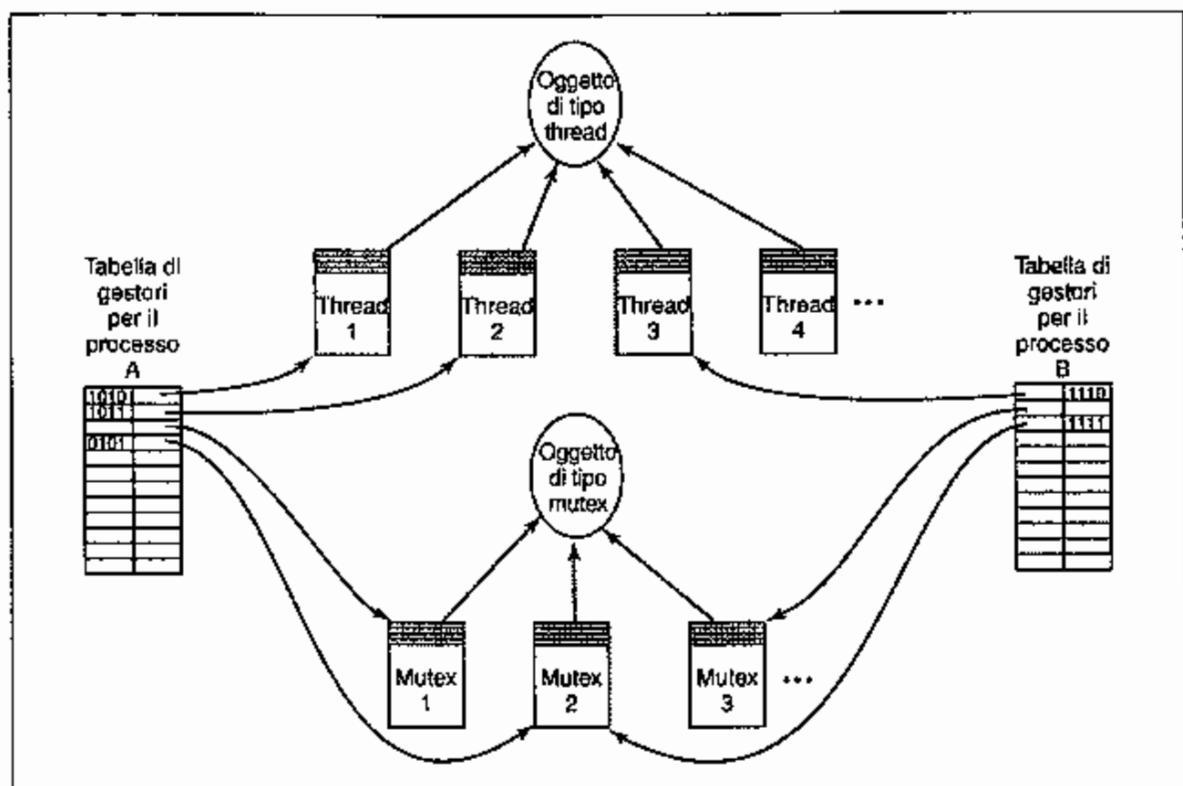


Figura 11.11 La relazione tra tabelle dei gestori, oggetti e tipi di oggetti.

per esempio, non appena il gestore di plug-and-play scopre dei dispositivi, esso crea un oggetto dispositivo per ognuno di loro, ed inserisce questi oggetti nello spazio dei nomi. Quando il sistema è completamente avviato, tutti i dispositivi di I/O, le partizioni di disco, ed altre importanti scoperte sono nello spazio dei nomi degli oggetti.

Non tutti gli oggetti vengono inseriti col metodo di Colombo – andare a guardare quello che si scopre; alcune componenti dell'esecutivo guardano nel registro per vedere cosa fare. Un esempio importante sono i driver di dispositivo: durante la fase di avvio, il sistema guarda nel registro per vedere quali sono i driver di dispositivi necessari. Una volta caricati ad uno ad uno, si crea un oggetto per ognuno, ed il suo nome viene inserito nello spazio degli oggetti. All'interno del sistema, il driver è riferito da un puntatore all'oggetto.

Sebbene lo spazio dei nomi degli oggetti sia cruciale per tutte le operazioni del sistema, poche persone sanno che esiste, perché non è visibile all'utente senza l'utilizzo di particolari strumenti di visualizzazione. Uno di questi strumenti è *winobj*, disponibile gratis al sito [www.sysinternals.com](http://www.sysinternals.com); quando viene mandato in esecuzione, esso mostra uno spazio dei nomi degli oggetti che tipicamente contiene le directory di oggetti mostrate in Figura 11.2, e poche altre.

La directory dal nome un po' strano \?? contiene i nomi di tutti i dispositivi in stile MS-DOS come A: per il floppy disk, C: per il primo disco rigido; questi nomi sono in effetti collegamenti simbolici alla directory \Device dove gli oggetti risiedono. Il nome \?? è stato scelto in modo che tale directory risulti la prima visualizzata in ordine alfabetico, per velocizzare la ricerca di pathname che iniziano con il nome del drive. Il contenuto delle altre directory degli oggetti è sostanzialmente auto-esplicativo.

### 11.3.3 Sottosistemi di ambiente

Ritornando alla Figura 11.7, osserviamo che Windows 2000 ha componenti in modo kernel e componenti in modo utente. Ora abbiamo completato il nostro esame delle componenti in modo kernel, ed è tempo di soffermarci sulla componenti modo utente, che sono di tre tipi: DLL, sottosistemi di ambiente e processi di servizio. Queste componenti lavorano insieme per fornire ad ogni processo utente un'interfaccia diversa dall'interfaccia delle chiamate di sistema per Windows 2000.

Windows 2000 supporta tre diverse API documentate: Win32, POSIX, e OS/2; ognuna di queste interfacce ha una lista pubblicata di chiamate di libreria che i programmati possono usare. Il lavoro delle DLL (Librerie dinamiche di collegamenti – Dynamic Link Library) e dei sottosistemi di ambiente è di implementare le funzionalità dell'interfaccia pubblicata, quindi di nascondere la vera interfaccia delle chiamate di sistema ai programmi applicativi. In particolare, l'interfaccia Win32 è l'interfaccia ufficiale di Windows 2000, Windows NT, Windows 95/98/Me e, in parte, di Windows CE. Usando le DLL ed il sottosistema di ambiente Win32, un programma può essere scritto per specifiche Win32 e può girare, senza modifiche, su tutte le versioni di Windows, anche se alcune chiamate di sistema non sono le stesse nei diversi sistemi.

Per vedere come sono implementate queste interfacce, si guardi a Win32. Un programma Win32, normalmente, contiene molte chiamate alla funzioni API Win32, per esempio, `CreateWindow`, `DrawMenuBar` e `OpenSemaphore`; ci sono migliaia di tali chiamate, e molti programmi le utilizzano in modo pesante. Una possibile implementazione potrebbe essere quella di collegare staticamente ogni programma Win32 con le procedure di libreria che esso utilizza; se così si facesse, ogni programma binario conterebbe nel suo codice una copia di ogni procedura che utilizza.

Il problema di questo approccio è lo spreco di memoria nel caso in cui l'utente abbia più programmi aperti allo stesso tempo, che usano le stesse procedure di libreria. Per esempio, *Word*, *Excel* e *Powerpoint*, utilizzano esattamente le stesse procedure per aprire finestre di dialogo, disegnare finestre, mostrare menu, gestire la clipboard, eccetera, cosicché, se un utente li avesse aperti tutti e tre in contemporanea, vi sarebbero in memoria tre copie identiche di ogni libreria.

Per risolvere questo problema, tutte le versioni di Windows supportano librerie con-

| Directory     | Contenuto                                                               |
|---------------|-------------------------------------------------------------------------|
| ??            | Il punto iniziale per recuperare i dispositivi MS-DOS come C:           |
| Device        | Tutti i dispositivi I/O scoperti                                        |
| Driver        | Oggetti corrispondenti ad ognuno dei driver di dispositivi caricati     |
| ObjectTypes   | Gli oggetti di tipo mostrati in Fig.11.11                               |
| Windows       | Oggetti per spedire messaggi a tutte le finestre                        |
| BaseNamedObjs | Oggetti creati da utenti come semafori, mutex, eccetera                 |
| ArchName      | I nomi delle partizioni scoperte dal caricatore di avvio                |
| NLS           | Gli oggetti per il supporto del linguaggio nazionale                    |
| FileSystem    | Oggetti di driver di file system e oggetti riconoscitori di file system |
| Security      | Oggetti appartenenti al sistema di sicurezza                            |
| KnownDLLs     | Librerie chiave condivise che sono aperte subito e tenute aperte        |

Figura 11.12 Alcune directory tipiche nello spazio dei nomi di oggetti.

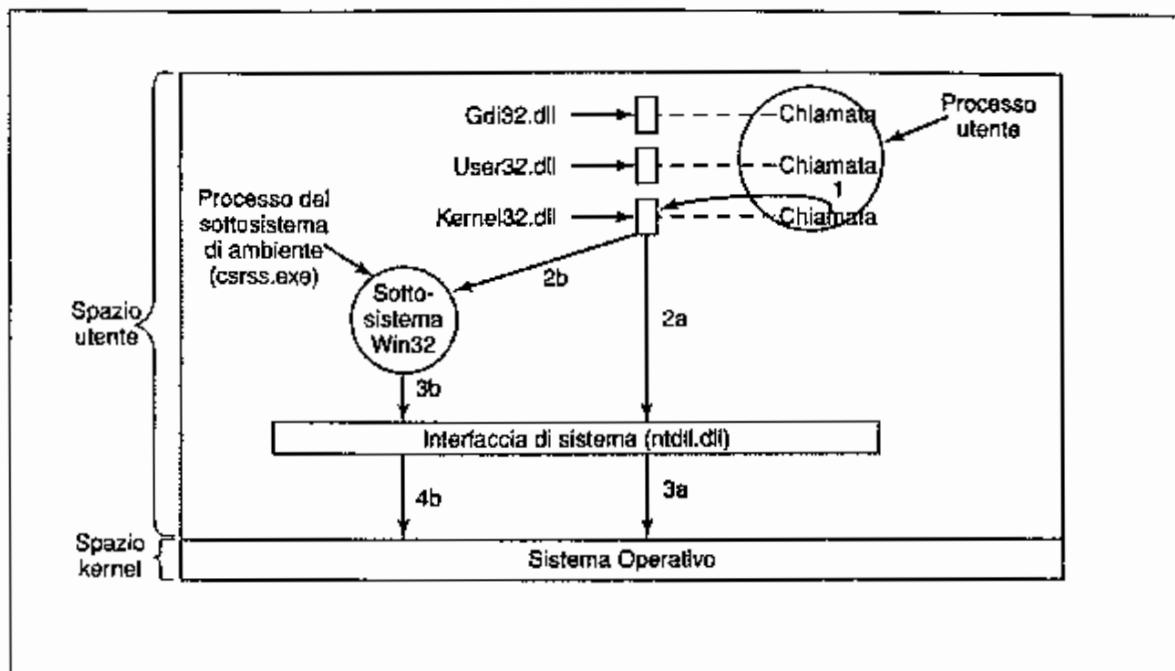


Figura 11.13 Varie strade per implementare le chiamate di funzioni Win32.

divise, chiamate **DLL** (Dynamic Link Library – Librerie dinamiche di collegamenti); ciascuna raccoglie in un unico file, di solito (ma non sempre) con estensione *.dll*, un insieme di procedure di librerie strettamente collegate, e le strutture dati relative. Quando una applicazione viene collegata, il linker vede che alcune procedure di libreria appartengono ad una qualche DLL, e memorizza queste informazioni nell'intestazione dell'eseguibile. Le chiamate a procedure nelle DLL sono fatte in modo indiretto, attraverso un vettore di trasferimento nello spazio degli indirizzi del chiamante; inizialmente, questo vettore è riempito con zeri, poiché gli indirizzi delle procedure da chiamare non sono ancora noti.

Quando parte un processo applicativo, si individuano le DLL necessarie (su disco o in memoria) e si mappano nello spazio degli indirizzi virtuale del processo. Il vettore di trasferimento viene quindi riempito con gli indirizzi corretti, per poter chiamare le procedure tramite tale vettore, con una trascurabile perdita di efficienza. Il vantaggio, in questo caso, è che basta una sola copia del testo della DLL nella memoria fisica per più programmi applicativi che hanno mappata la stessa DLL (ma ogni processo possiede la propria copia dei dati privati statici della DLL). Windows 2000 usa le DLL in modo estremamente pesante in tutti gli aspetti del sistema.

Adesso abbiamo abbastanza nozioni di base per vedere come vengono implementate Win32 e altre interfacce di processi. Ogni processo utente, di solito, è collegato con un certo numero di DLL che insieme implementano l'interfaccia Win32; per fare una chiamata API, si chiama una di queste procedure nelle DLL, come mostrato nel Passo 1 nella Figura 11.13. Cosa avviene dopo dipende dalla chiamata API Win32, chiamate differenti sono implementate in modo diverso.

In alcuni casi, una DLL chiama un'altra DLL (*ntdll.dll*) che di fatto esegue una trap al sistema operativo; questo cammino è mostrato come Passo 2a e 3a in Figura 11.13. È anche possibile che una DLL faccia tutto il lavoro da sola senza fare alcuna chiamata di sistema. Per altre chiamate API Win32 vengono intraprese strade differenti, ad esempio,

| File         | Modo   | Funz. | Contenuto                                                                                                         |
|--------------|--------|-------|-------------------------------------------------------------------------------------------------------------------|
| hal.dll      | Kernel | 95    | Gestione dell'hardware a basso livello, ad esempio la porta I/O                                                   |
| ntoskrnl.exe | Kernel | 1209  | Sistema operativo Windows 2000 (kernel + esecutivo)                                                               |
| win32k.sys   | Kernel | -     | Molte chiamate di sistema comprese quasi tutte quelle dedicate alla grafica                                       |
| ntdll.dll    | Utente | 1179  | Consente di passare da modo utente a modo kernel                                                                  |
| csrss.exe    | Utente | 0     | Processo di sottosistema di ambiente Win32                                                                        |
| kernel32.dll | Utente | 823   | Il nucleo delle chiamate di sistema (non grafiche)                                                                |
| gdi32.dll    | Utente | 543   | Chiamate per i caratteri, il testo, i colori, le bitmap, i disegni, la penna, il pennello, la tavolozza, eccetera |
| user32.dll   | Utente | 695   | Chiamate per finestre, icone, menu, cursore, dialog, clipboard, eccetera                                          |
| advapi32.dll | Utente | 557   | Chiamate di gestione di registro, di crittografia, di sicurezza                                                   |

**Figura 11.14** Alcuni file chiave di Windows 2000, il modo in cui sono eseguiti, il numero di chiamate di funzioni esportate, ed il principale contenuto di ogni file. Le chiamate in win32k.sys non sono formalmente esportate perché win32k.sys non viene chiamata in modo diretto.

prima si spedisce un messaggio al processo di sottosistema Win32 (csrss.exe), che dopo aver fatto un po' di lavoro fa una chiamata di sistema (passi 2b, 3b, 4b). Anche qui, in alcuni casi, il sottosistema di ambiente fa tutto il lavoro nello spazio utente e restituisce tutto immediatamente. Per aumentare le prestazioni, il passaggio del messaggio tra il processo applicativo ed il processo di sottosistema Win32 viene ottimizzato mediante un meccanismo di chiamata di procedura locale (LPC) implementata dall'esecutivo e mostrato in Figura 11.7.

Nella prima versione di Windows NT, praticamente tutte le chiamate API Win32 prendevano il cammino 2b, 3b, 4b portando un pezzo considerevole di sistema operativo nello spazio utente (ad esempio, la grafica). In ogni caso, a partire da NT 4.0, per ragioni di prestazioni molto codice è stato spostato in modo utente (nel driver Win32/GDI in Figura 11.7). In Windows 2000, solo un piccolo numero di chiamate API Win32, per esempio la creazione di processi e thread, prende la strada più lunga, le altre prendono una strada diretta, evitando il sottosistema di ambiente Win32.

In Figura 11.13, sono mostrate le tre DLL più importanti, ma ne esistono altre: in effetti ne esistono 800 diverse, nella directory \winnt\system32 per un totale di 130MB. Per evitare della confusione, il numero di file DLL è oltre 800, mentre il numero di chiamate API in esse contenute supera le 13000 (i 29 milioni di righe di codice compilato dovranno pur produrre qualcosa, dopotutto). In Figura 11.14 vengono elencate alcune delle più importanti DLL: viene fornito anche il rispettivo numero di funzioni esportate (cioè quelle visibili al di fuori del file), numero che tende a variare (leggi aumentare) nel tempo. Il numero di funzioni esportate nella prima release di ntdll.dll in Windows 2000 è 1179: queste sono vere chiamate di sistema; le 1209 chiamate esportate da ntoskrnl.exe sono funzioni disponibili per i driver di dispositivo ed altro codice collegato con il kernel. Utilizzando il programma *depends* nel kit SDK si può osservare la lista di tutte le funzioni esportate da ogni file .exe o .dll.

Sebbene l'interfaccia per i processi Win32 sia la più importante, vi sono anche le altre due: POSIX e OS/2. L'ambiente POSIX fornisce un supporto minimo ad applicazioni UNIX, infatti supporta solo la funzionalità P1003.1 e poco altro; non possiede thread,

sistemi a finestra, e rete, per esempio. In pratica, far migrare un qualsiasi programma UNIX a Windows 2000 usando questo sottosistema è praticamente impossibile; è stato incluso solo perché il Governo degli Stati Uniti richiedeva che il sistema operativo supportasse in qualche modo POSIX. Questo sottosistema non è autocontenuto e usa fortemente il sottosistema Win32, ma senza esportare l'intera interfaccia Win32 ai suoi programmi utente (il che sarebbe stata una cosa utile, ed a costo zero per Microsoft).

Per permettere agli utenti UNIX di migrare a Windows 2000, Microsoft ha un prodotto chiamato Interix che fornisce un maggior grado di compatibilità con UNIX rispetto al sottosistema POSIX.

Il sottosistema OS/2, è anch'esso limitato in funzionalità e non supporta nessuna applicazione grafica; in pratica è completamente inutile. Quindi l'idea originale di avere diverse interfacce del sistema operativo implementate da diversi processi nello spazio utente è essenzialmente andata persa; quello che rimane è una implementazione completa di Win32 in modo kernel, e poco altro.

## 11.4.1 processi e i thread in Windows 2000

Windows 2000 possiede alcuni concetti per gestire la CPU e raggruppare le risorse. Qui di seguito li esamineremo, discutendo di alcune chiamate API Win32 importanti, e mostreremo come si implementano questi concetti.

### 11.4.1 Concetti di base

Windows 2000 supporta i processi tradizionali, che possono comunicare e sincronizzarsi l'uno con l'altro, esattamente come succede in UNIX; ogni processo contiene almeno un thread, che contiene a sua volta almeno un thread leggero (fiber). In più, i processi possono essere raccolti in job per motivi di gestione delle risorse. Job, thread, processi e thread leggeri sono un insieme generale di strumenti per gestire le risorse ed il parallelismo, sia su un uniprocessore (macchine a CPU singola) sia su multiprocessori (macchine con molte CPU); un breve riassunto di questi quattro concetti è mostrato in Figura 11.15.

Esaminiamo questi concetti dal più ampio al più piccolo. Un job (lavoro) in Windows 2000 è una collezione di uno o più processi che devono essere gestiti come una unità; in particolare, vi sono limiti di quote e risorse associati ad ogni job, memorizzati nel corrispondente oggetto job. Le quote comprendono ad esempio il numero massimo di processi (per impedire che un processo generi un numero illimitato di figli), il tempo totale di CPU disponibile per ogni processo individualmente, e per tutti i pro-

| Nome           | Descrizione                                                       |
|----------------|-------------------------------------------------------------------|
| Job            | Collezione di processi che condividono quota e limiti             |
| Processo       | Contenitore di risorse possedute                                  |
| Thread         | Entità schedulata dal kernel                                      |
| Thread leggero | Thread di minor peso gestito nella sua interezza in spazio utente |

Figura 11.15 I concetti di base usati per la gestione della CPU e delle risorse.

cessi in totale, ed il massimo utilizzo di memoria per processo ed in totale. I job possono imporre anche restrizioni di sicurezza sui processi in essi contenuti, come il fatto di non poter acquisire potere di amministratore (superuser), anche con password inserita correttamente.

I processi sono più interessanti dei job ed anche più importanti; come in UNIX, i processi sono contenitori di risorse. Ogni processo ha uno spazio degli indirizzi di 4GB, con l'utente che occupa i 2GB in fondo (in modo opzionale 3GB, nelle versioni Server Avanzato e DataCenter), ed il sistema operativo occupa il resto; perciò il sistema operativo è presente in ogni spazio di indirizzo dei processi, sebbene protetto dalle intrusioni dall'unità hardware per la gestione della memoria. Un processo ha un ID di processo, uno o più thread, una lista di handle (gestori – gestiti in modo kernel), ed un token di accesso che mantiene le informazioni di sicurezza. I processi sono creati utilizzando una chiamata Win32 che prende come input il nome di un file eseguibile, definisce il contenuto iniziale dello spazio degli indirizzi e crea il primo thread.

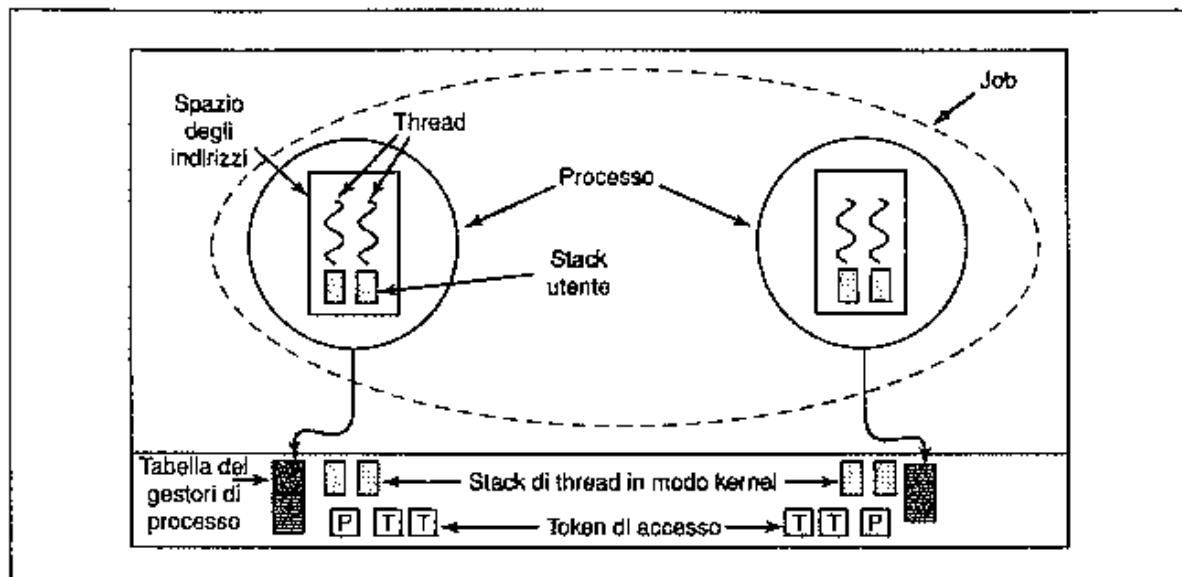
Ogni processo inizia con un thread, ma se ne possono creare di nuovi dinamicamente. I thread formano la base per lo scheduling della CPU, poiché il sistema operativo sceglie sempre un thread da far girare (e non un processo); di conseguenza, ogni thread ha uno stato (pronto, in esecuzione, bloccato, eccetera) mentre un processo non ha stati. I thread possono essere creati dinamicamente da una chiamata Win32 che specifica l'indirizzo dal quale incominciare tra quelli presenti nello spazio degli indirizzi del processo. Ogni thread ha un thread ID, che viene preso dallo stesso spazio di indirizzi dell'ID del processo; per questo, un ID non può essere mai utilizzato da un processo e da un thread nello stesso momento. Gli ID del processo e del thread sono multipli di quattro, in modo da poter essere usati come indici di byte nelle tabelle del kernel, come gli altri oggetti.

Un thread normalmente gira in modo utente, ma quando fa una chiamata di sistema passa in modo kernel, e continua ad essere eseguito come thread con le stesse proprietà e gli stessi limiti che aveva in modo utente. Ogni thread possiede due stack, uno da usare quando si trova in modo utente e l'altro quando si trova in modo kernel; oltre allo stato, l'ID, e i due stack, ogni thread ha un contesto (nel quale salvare i suoi registri quando non è in esecuzione), un'area privata per le sue variabili locali ed eventualmente, un token di accesso. Se è provvisto di token di accesso, questo sovrascrive il token di accesso del processo, in modo da permettere ai thread-client di passare i propri diritti di accesso ai thread-server che stanno lavorando per loro. Quando il thread smette di essere eseguito, può uscire, e quando esce l'ultimo thread attivo nel processo, il processo termina.

È importante capire che i thread sono un concetto di scheduling, non un concetto di proprietà di risorsa; ogni thread è in grado di accedere a tutti gli oggetti che appartengono al suo processo, tutto quello che deve fare è prendere il gestore e fare una chiamata Win32 adeguata. Il thread non ha restrizioni di accesso ad oggetti che sono stati creati o aperti da altri thread; dopo che un gestore di oggetto è posto nella tabella dei gestori del processo, ogni altro thread nel processo può utilizzarlo.

Oltre ai thread normali in esecuzione all'interno dei processi utente, Windows 2000 ha un certo numero di thread demoni che girano solo nello spazio kernel, e non vengono associati con nessun processo utente (essi sono associati a processi speciali di sistema o al processo inattivo). Alcuni eseguono compiti amministrativi, come quello di scrivere pagine sporche sul disco, mentre altri formano un pool che può essere assegnato ad una componente dell'esecutivo o di un driver che ha bisogno di svolgere un po' di lavoro in modo asincrono ed in background. Studieremo alcuni di questi thread più avanti quando arriveremo alla gestione della memoria.

L'operazione di commutazione (switching) dei thread in Windows 2000 è relativa-



**Figura 11.16** La relazione tra job, processi e thread. Parecchi thread leggeri possono anche essere multiplexati in un thread (non mostrato).

mente costosa perché fare lo switch di un thread richiede l'ingresso e l'uscita dal modo kernel, quindi, per fornire una forma di pseudo-parallelismo molto blanda, Windows 2000 fornisce i **thread leggeri** (*fiber*, letteralmente fibra), che sono un po' come thread ma vengono schedulati nello spazio utente dal programma che li ha creati (o dal suo sistema run-time). Ogni thread può avere molti thread leggeri, nello stesso modo in cui un processo può avere più thread, la differenza è il caso in cui un thread leggero si blocca in modo logico: esso si mette in una coda di thread leggeri bloccati e seleziona un altro thread leggero da eseguire nel contesto del suo thread.

Il sistema operativo non si cura di questa transizione perché il thread rimane in esecuzione, anche se prima era in esecuzione un thread leggero e poi un altro. In effetti, il sistema operativo non sa nulla dei thread leggeri, e infatti non vi sono oggetti esecutivi associati ai thread leggeri, mentre ve ne sono per i job, i processi ed i thread, né esistono vere chiamate di sistema per i thread leggeri. Ad ogni modo, vi sono chiamate API Win32, che sono tra quelle che non eseguono chiamate di sistema, citate nel corso della trattazione della Figura 11.13. La relazione tra i job, i processi ed i thread è mostrata in Figura 11.16.

Sebbene non ne discuteremo in dettaglio, Windows 2000 è in grado di andare in esecuzione su un sistema multiprocessore simmetrico; questo vuol dire che il codice del sistema operativo deve essere rientrante in modo completo, cioè, ogni procedura deve essere scritta in modo che due o più CPU possano cambiare le proprie variabili allo stesso tempo e senza causare problemi. In molti casi questo vuol dire che le sezioni di codice devono essere protette da spin lock o mutex per mantenere le altre CPU ferme fino a quando la prima non ha finito (cioè un accesso seriale alle regioni critiche). Il numero di CPU che il sistema è in grado di manipolare è limitato da restrizioni di licenza ed è mostrato in Figura 11.2; non vi è alcuna ragione tecnica per la quale Windows Professional non può girare su un multiprocessore a 32 nodi – dopo tutto, esso è di fatto lo stesso binario del Server Datacenter.

Il limite superiore di 32 CPU è un limite serio perché si usano mappe di bit della dimensione di una parola (word) per tenere traccia dell'utilizzo delle CPU in vario modo; per esempio, una mappa di bit della dimensione di una parola tiene traccia di quali delle (al massimo) 32 CPU sono attualmente in stato inattivo, ed un'altra mappa viene usa-

ta per ogni processo, in modo da avere la lista delle CPU sulle quali può girare. La versione a 64-bit di Windows 2000 dovrebbe essere in grado di supportare senza problemi fino a 64 CPU; però occorrerebbe cambiare il codice in modo sostanziale (usare più parole per le mappe) per andare oltre.

## 11.4.2 Chiamate API per la gestione di job, processi, thread e thread leggeri

I nuovi processi sono creati chiamando la funzione API Win32 `CreateProcess`. Questa funzione ha 10 parametri, ognuno dei quali ha molte opzioni. Il progetto è chiaramente molto più complicato dello schema UNIX, nel quale la funzione `fork` non ha parametri e la `exec` ne ha appena tre: il puntatore al nome del file da eseguire, l'array di parametri della linea di comando (già analizzata sintatticamente), le stringhe di ambiente. Grossso modo, i 10 parametri della `CreateProcess` sono:

1. Un puntatore al nome del file da eseguire.
2. La linea di comando (non analizzata sintatticamente).
3. Un puntatore al descrittore di sicurezza per il processo.
4. Un puntatore al descrittore di sicurezza per il thread iniziale.
5. Un bit che dice se il processo eredita dei gestori di oggetti dal creatore.
6. Un insieme di generici flag (ad esempio, il modo errore, priorità, debugging, console).
7. Un puntatore alle stringhe di ambiente.
8. Un puntatore al nome della directory corrente del nuovo processo.
9. Un puntatore ad una struttura che descrive la finestra iniziale sullo schermo.
10. Un puntatore ad una struttura che restituisce 18 valori al chiamante.

Windows 2000 non impone alcun tipo di gerarchia padre-figlio o altro; tutti i processi sono creati uguali (nessun processo viene creato più uguale di altri). Comunque dato che uno dei 18 parametri restituiti al processo creatore è un gestore del nuovo processo (e gli consente un notevole controllo sul nuovo processo), vi è una gerarchia implicita in termini di chi ha il gestore di chi. Sebbene questi gestori non possano essere semplicemente passati ad altri processi, vi è un modo per cui un processo può duplicare un gestore utile per un altro processo e quindi fornirglielo, in modo che la gerarchia di gestori implicita non duri a lungo.

Ogni processo in Windows 2000 viene creato con un solo thread, ma può creare successivamente più thread; la creazione di thread è più semplice della creazione di un processo: `CreateThread` possiede solo 6 parametri invece di 10:

1. Il descrittore di sicurezza (opzionale).
2. La dimensione iniziale dello stack.
3. L'indirizzo di partenza.
4. Un parametro definito dall'utente.
5. Lo stato iniziale del thread (pronto o bloccato).
6. L'ID relativo.

Il kernel effettua la creazione del thread, quindi se ne occupa veramente (cioè essi non sono implementati semplicemente nello spazio utente, come in altri sistemi).

## Comunicazione tra i processi

I thread possono comunicare in una grande varietà di modi, tra cui le pipe, le pipe con nome (named pipe), i mailslot, i socket, le chiamate di procedure remote e i file condivisi. Le pipe hanno due modi: a byte o a messaggi: il modo è selezionato al momento della creazione. Le pipe a byte lavorano come in UNIX; le pipe a messaggi sono simili, ma conservano i limiti del messaggio, ovvero quattro scritture di 128 byte saranno lette come quattro messaggi di 128 byte e non come un messaggio di 512 byte, come potrebbe accadere in modalità a byte. Esistono anche le pipe con nome e hanno tutti e due i modi come le pipe regolari; esse possono anche essere usate in rete; le pipe regolari no.

I mailslot sono funzionalità di Windows 2000 non presenti in UNIX; essi sono simili alle pipe, ma non del tutto. In primis, sono unidirezionali, mentre le pipe sono bidirezionali; possono essere usati in rete ma non forniscono consegne garantite; infine, ammettono che il processo mittente spedisca il messaggio a molti destinatari (broadcast), invece che ad uno solo.

I socket sono come le pipe, fatta eccezione per il fatto che normalmente connettono processi appartenenti a macchine diverse; per esempio, un processo scrive su un socket ed un altro su una macchina remota legge da questo. I socket possono anche essere adoperati per connettere processi sulla stessa macchina ma, dato che comportano più carico delle pipe, sono impiegati in genere solo nel contesto di rete.

Le chiamate remote di procedura sono il modo con cui un processo A chiede ad un processo B di chiamare una procedura, nello spazio degli indirizzi di B, per conto di A, e restituire il risultato ad A. Esistono varie restrizioni sui parametri, per esempio, non ha senso passare un puntatore a processi diversi.

Infine, i processi possono condividere della memoria mappando lo stesso file nello stesso momento; così, tutte le scritture fatte da un processo appaiono negli spazi degli indirizzi degli altri processi. Usando questo meccanismo, si può implementare facilmente il buffer condiviso utilizzato nei problemi produttore-consumatore.

Dato che Windows 2000 possiede numerosi meccanismi di comunicazione tra processi, analogamente fornisce numerosi meccanismi di sincronizzazione, tra cui semafori, mutex, regioni critiche ed eventi. Tutti questi meccanismi funzionano sui thread e non sui processi, perciò se un thread si blocca ad un semaforo, altri thread nel processo (se esistono) non sono influenzati e possono continuare l'esecuzione.

Un semaforo viene creato utilizzando la funzione API CreateSemaphore, che può inizializzarlo ad un certo valore e definire il massimo valore possibile. I semafori sono oggetti del kernel, quindi hanno descrittori di sicurezza e gestione; il gestore di un semaforo può essere duplicato usando la funzione OpenSemaphore e può essere passato ad un nuovo processo, in modo che molti processi possano sincronizzarsi sullo stesso semaforo. Ci sono le chiamate up e down, ma hanno dei nomi particolari: ReleaseSemaphore (up), WaitForSingleObject (down). È anche possibile fornire alla chiamata WaitForSingleObject un timeout in modo che il thread chiamante possa essere rilasciato anche se il semaforo rimane a 0 (ma questo timer reintroduce le corse critiche).

I mutex sono anch'essi oggetti di kernel utilizzati per la sincronizzazione, ma più semplici dei semafori poiché non hanno contatori. Sono essenzialmente lock, con funzioni API per bloccare (WaitForSingleObject) e per sbloccare (ReleaseMutex); come per i gestori di semafori, i gestori di mutex possono essere duplicati e passati tra processi in modo che più thread in processi diversi possano accedere allo stesso mutex.

Il terzo meccanismo di sincronizzazione è basato sulle **sezioni critiche** (che in altri punti di questo libro sono state chiamate **regioni critiche**), le quali sono molto simili ai mutex, tranne che sono locali allo spazio degli indirizzi del thread che li crea. Poiché le sezioni critiche non sono oggetti di kernel, esse non hanno gestori o descrittori di sicu-

rezza, e non possono essere passate tra processi. Le operazioni di blocco e sblocco vengono fatte con `EnterCriticalSection` e `LeaveCriticalSection`, rispettivamente; poiché queste funzioni API sono eseguite inizialmente nello spazio utente e fanno chiamate al kernel solo quando è necessario un blocco, esse sono più veloci dei mutex.

L'ultimo meccanismo di sincronizzazione usa degli oggetti di kernel chiamati eventi, che possono essere di due tipi: **eventi ad azzeramento manuale (manual-reset events)** e **eventi ad azzeramento automatico (auto-reset events)**; ogni evento può essere in uno dei due stati seguenti: fissato e liberato. Un thread può attendere l'accadere di un evento con `WaitForSingleObject`; e se un altro thread segnala un evento con `SetEvent`, quello che succede dipende dal tipo di evento. Se era un evento ad azzeramento manuale, vengono liberati tutti i thread in attesa, e l'evento rimane fissato, finché non viene cambiato manualmente allo stato liberato con `ResetEvent`. Se era un evento ad azzeramento automatico, e se vi sono uno o più thread che stanno aspettando, si libera esattamente un thread e l'evento passa in stato liberato, e se poi avviene un `SetEvent` mentre nessun thread è in attesa, questo fatto viene ricordato lasciando l'evento allo stato fissato, in modo da poter liberare immediatamente il primo thread che si metterà in attesa.

Gli eventi, i mutex ed i semafori possono tutti avere un nome ed essere memorizzati nel file system, come le pipe con nome; due o più processi possono sincronizzarsi prendendo lo stesso evento, mutex o semaforo; oppure, uno dei processi crea un oggetto e dopo produce gestori duplicati per gli altri; entrambi gli approcci sono sempre possibili.

| Funzione API Win32                  | Descrizione                                                       |
|-------------------------------------|-------------------------------------------------------------------|
| <code>CreateProcess</code>          | Crea un nuovo processo                                            |
| <code>CreateThread</code>           | Crea un nuovo thread in un processo esistente                     |
| <code>CreateFiber</code>            | Crea un nuovo thread leggero                                      |
| <code>ExitProcess</code>            | Termina il processo attivo e tutti i suoi thread                  |
| <code>ExitThread</code>             | Termina questo thread                                             |
| <code>ExitFiber</code>              | Termina questo thread leggero                                     |
| <code>SetPriorityClass</code>       | Definisce la classe di priorità per un processo                   |
| <code>SetThreadPriority</code>      | Definisce la priorità per un thread                               |
| <code>CreateSemaphore</code>        | Crea un nuovo semaforo                                            |
| <code>CreateMutex</code>            | Crea un nuovo mutex                                               |
| <code>OpenSemaphore</code>          | Apre un semaforo esistente                                        |
| <code>OpenMutex</code>              | Apre un mutex esistente                                           |
| <code>WaitForSingleObject</code>    | Blocca su un singolo semaforo, mutex, eccetera                    |
| <code>WaitForMultipleObjects</code> | Blocca su un insieme di oggetti di cui si possiedono i gestori    |
| <code>PulseEvent</code>             | Definire il valore di un evento a segnalato e poi a non segnalato |
| <code>ReleaseMutex</code>           | Rilascia un mutex per permettere che un altro thread lo prenda    |
| <code>ReleaseSemaphore</code>       | Aumenta il valore del semaforo di uno                             |
| <code>EnterCriticalSection</code>   | Acquisisce un lock su una regione critica                         |
| <code>LeaveCriticalSection</code>   | Rilascia un lock su una regione critica                           |

Figura 11.17 Alcune delle chiamate Win32 per gestire processi, thread e thread leggeri.

Il numero di chiamate API Win32 che hanno a che fare con processi, thread e thread leggeri è intorno a 100, ed un numero considerevole di esse ha a che fare con la comunicazione tra processi in una forma o in un'altra; un riassunto delle chiamate discusse fino a qui e di altre ancora si può trovare in Figura 11.17.

La maggior parte delle chiamate in Figura 11.17 sono state discusse o sono autoesplicative; si noti ancora che non tutte le funzioni sono chiamate di sistema. Come abbiamo già detto, Windows 2000 non conosce i thread leggeri, che sono interamente implementati nello spazio utente. Di conseguenza, la chiamata `CreateFiber` fa il proprio lavoro interamente nello spazio utente senza fare alcune chiamate di sistema (a meno che non debba allocare della memoria). Molte altre chiamate Win32 hanno questa proprietà, tra cui `EnterCriticalSection` e `LeaveCriticalSection`, come si è già precedentemente sottolineato.

### 11.4.3 Implementazione dei processi e dei thread

I processi ed i thread sono più importanti e più elaborati dei job e dei thread leggeri, quindi ci concentreremo maggiormente sui primi due. Un processo viene creato quando un altro processo fa una chiamata Win32 `CreateProcess`, la quale chiama una procedura (modo utente) in `kernel32.dll` che crea il processo attraverso vari passi, usando diverse chiamate di sistema ed altro ancora.

1. Il file eseguibile fornito come parametro viene esaminato ed aperto; se è un file valido, POSIX, OS/2, Windows 16-bit, o MS-DOS, si prepara per esso un ambiente speciale; se è un file .exe a 32-bit Win32 valido, si controlla il registro per vedere se è speciale in un qualche modo (ad esempio, se deve essere eseguito sotto la supervisione di un debugger). Tutto questo viene fatto in modo utente dentro `kernel32.dll`.
2. Si fa una chiamata di sistema, `NtCreateProcess`, per creare un oggetto processo vuoto, ed inserirlo dentro lo spazio dei nomi del gestore di oggetti; inoltre, il gestore di oggetti crea un blocco di controllo di processi per l'oggetto e lo inizializza con l'ID del processo, le quote, il token di accesso e vari altri campi. Inoltre si crea anche un oggetto sezione, per tenere traccia dello spazio degli indirizzi del processo.
3. Quando `kernel32.dll` riprende il controllo, fa un'altra chiamata di sistema, `NtCreateThread`, per creare il thread iniziale. Vengono creati anche gli stack dell'utente e del kernel; la dimensione degli stack viene presa dall'intestazione del file eseguibile.
4. `kernel32.dll` manda quindi un messaggio al sottosistema di ambiente Win32 comunicando che esiste il nuovo processo, e passandogli i gestori del processo e del thread. Il processo ed i thread sono inseriti nelle tabelle del sottosistema, in modo che esso abbia una lista completa di tutti i processi e di tutti i thread. Il sottosistema, quindi, visualizza un cursore a forma di clessidra per comunicare all'utente che qualcosa sta accedendo, ma che il cursore nel frattempo può essere utilizzato. Quando un processo fa la sua prima chiamata alla GUI, solitamente crea una finestra ed il cursore viene rimosso (scade dopo due secondi se non vi sono chiamate successive).
5. A questo punto il thread è in grado di andare in esecuzione; incomincia eseguendo una procedura di sistema a tempo di esecuzione per completare l'inizializzazione.
6. La procedura a tempo di esecuzione fissa la priorità del thread, comunica alle DLL caricate che è presente il nuovo thread, e fa altri piccoli lavori. Infine, incomincia a far eseguire il codice del programma principale del processo.

Anche la creazione di un thread si compone di un certo numero di passi, ma non vi entreremo troppo in dettaglio; essa inizia quando il processo in esecuzione effettua una `CreateThread`, che chiama una procedura dentro la `kernel32.dll`. Questa procedura alloca uno stack utente nel processo chiamante e successivamente chiama `NtCreateThread` per creare un oggetto `thread` nell'esecutivo, inizializzarlo, e creare ed inizializzare anche un blocco di controllo di `thread`. Nuovamente, il sottosistema Win32 viene avvisato e immette il nuovo `thread` nelle sue tabelle; in seguito, il `thread` parte e completa la propria inizializzazione.

Quando si crea un processo o un `thread`, viene restituito il loro gestore; tale gestore può essere utilizzato per iniziare, fermare, uccidere o esaminare il processo o il `thread`. Il proprietario del gestore può passarlo ad un altro processo in modo sicuro e controllato; questa tecnica è usata per permettere ai debugger di possedere il controllo sui processi da controllare.

## Scheduling

Windows 2000 non possiede un `thread` centrale per la schedulazione; al contrario, quando un `thread` non può continuare il proprio svolgimento, entra in modo kernel e attiva lo scheduler al proprio interno per vedere con quale altro `thread` è possibile far cambio. Lo scheduler può essere chiamato se si verifica una delle seguenti condizioni:

- 1. Il `thread` si blocca su un semaforo, mutex, evento, I/O, eccetera.
- 2. Il `thread` comunica con un oggetto (ad esempio fa un up su un semaforo).
- 3. Il quanto di esecuzione del `thread` scade.

Nel caso 1, il `thread` è già in esecuzione in modo kernel per attuare una operazione sul gestore o sull'oggetto di I/O, ma non può continuare, e per questo deve salvare il proprio contesto, eseguire il codice dello scheduler per recuperare il suo successore, e caricare il contesto di quel `thread` per farlo incominciare.

Nel caso 2, il `thread` in esecuzione è di nuovo in modo kernel, ma dopo la segnalazione all'oggetto, può continuare dato che la segnalazione ad un oggetto non comporta mai un blocco. Ciò nonostante, il `thread` deve eseguire lo scheduler per controllare che il risultato della sua azione non abbia, in qualche modo, liberato un `thread` con priorità più alta, che adesso deve essere eseguito. Se questo è accaduto, avviene una comutazione di `thread`, dato che Windows 2000 è completamente prelasciabile (cioè, lo scambio di `thread` può avvenire in ogni momento, non solo alla fine del quanto del `thread` attivo).

Nel caso 3, avviene un'interruzione software al modo kernel, ed in quel momento il `thread` esegue il codice dello scheduler per vedere chi deve essere eseguito dopo di lui; a seconda di quali `thread` stanno aspettando, può essere selezionato lo stesso `thread`, nel qual caso esso riceve un nuovo quanto e continua l'esecuzione; altrimenti avviene uno scambio di `thread`.

Inoltre lo scheduler viene interpellato nei casi seguenti:

- 1. Quando viene completata una operazione di I/O
- 2. Quando scade una attesa a tempo.

Nel primo caso, un `thread` poteva essere in attesa dell'evento, e quindi viene subito rilasciato per essere eseguito; occorre controllare se deve prelazionare il `thread` in esecuzione, poiché non esiste nessun tempo minimo di esecuzione garantito. Lo scheduler non è

attivato dal gestore di interruzione (poiché questo dovrebbe mantenere le interruzioni inattive per troppo tempo); invece, mette in coda un DPC, per poco tempo, finché il gestore di interruzioni non termina. Nel secondo caso, un thread ha fatto un down su un semaforo o si è bloccato su qualche altro oggetto, ma il suo timeout è scaduto; è necessario che anche stavolta il gestore di interruzioni metta in coda un DPC per evitare che esso sia in esecuzione mentre lo è il gestore di interruzioni del clock. Se un thread è diventato pronto (ready) grazie a questo timeout, lo scheduler andrà in esecuzione e se nessun altro thread più importante è disponibile, il DPC verrà eseguito di seguito.

Eccoci adesso all'algoritmo di scheduling vero e proprio. L'API Win32 fornisce ai processi due meccanismi per influenzare lo scheduling dei thread; essi determinano in gran parte l'algoritmo. Per prima cosa, si ha una chiamata `SetPriorityClass` che definisce la classe di priorità per ogni thread nel processo chiamante; i valori permessi sono: real-time, alta, sopra il normale, normale, sotto il normale, e inattivo.

In secondo luogo, c'è la chiamata `SetThreadPriority` che definisce la priorità relativa per un certo thread (possibilmente, ma non obbligatoriamente, quella del thread chiamante) in relazione ad altri thread nello stesso processo. I valori permessi sono: critico sul tempo, il più alto, sopra il normale, normale, sotto il normale, il più basso, e inattivo. Con sei classi di processi e sette classi di thread, un thread può avere una delle 42 combinazioni possibili, che sarà l'input dell'algoritmo di scheduling.

Lo scheduler lavora come segue: il sistema ha 32 priorità, numerate da 0 a 31, e le 42 combinazioni sono mappate sulle 32 classi secondo la tabella di Figura 11.18. Il numero nella tabella determina la priorità di base del thread, ma in più, ogni thread ha la priorità corrente, che può essere più alta (ma non più bassa) della priorità di base e della quale discuteremo in breve.

Per utilizzare queste priorità nello scheduling, il sistema mantiene un array di 32 campi, corrispondenti alle priorità da 0 a 31 derivate dalla tabella di Figura 11.18; ogni campo dell'array punta alla testa di una lista di thread pronti alla priorità corrispondente. L'algoritmo di scheduling di base cerca nell'array dalla priorità 31 alla priorità 0; e non appena trova un campo non vuoto, seleziona il thread in testa alla coda, che viene eseguito per un quanto. Se il quanto scade, il thread viene posto alla fine della coda al suo livello di priorità, e il thread in testa viene scelto come thread da eseguire. In altre parole, quando vi sono più thread pronti al più alto livello di priorità, essi eseguono un Round Robin, ognuno per un quanto; se non esistono thread pronti, si esegue il thread inattivo.

Occorre notare che la procedura di scheduling viene attivata scegliendo un thread, a prescindere dal processo a cui appartiene, quindi lo scheduler non sceglie prima un pro-

|                          |                  | Priorità di classe di processi Win32 |      |                  |         |                  |           |
|--------------------------|------------------|--------------------------------------|------|------------------|---------|------------------|-----------|
|                          |                  | Tempo reale                          | Alto | Sopra il normale | Normale | Sotto il normale | In attesa |
| Priorità di thread Win32 | Critica          | 31                                   | 15   | 15               | 15      | 15               | 15        |
|                          | Più alto         | 26                                   | 15   | 12               | 10      | 8                | 6         |
|                          | Sopra al normale | 25                                   | 14   | 11               | 9       | 7                | 5         |
|                          | Normale          | 24                                   | 13   | 10               | 8       | 6                | 4         |
|                          | Sotto il normale | 23                                   | 12   | 9                | 7       | 5                | 3         |
|                          | Più basso        | 22                                   | 11   | 8                | 6       | 4                | 2         |
|                          | In attesa        | 16                                   | 1    | 1                | 1       | 1                | 1         |

Figura 11.18 La mappa dalle priorità Win32 alle priorità Windows 2000.

cesso e poi un thread in quel processo, ma osserva solo i thread, e non conosce neanche quale thread appartiene a quale processo. Su un multiprocessore, ogni CPU attua lo scheduling utilizzando l'array di priorità; uno spin lock viene utilizzato per garantire che in ogni istante solo una CPU stia ispezionando l'array.

L'array dei puntatori alle code è mostrato in Figura 11.19. La figura mostra che vi sono di fatto quattro categorie di priorità: realtime, utente, zero e inattivo, che è di fatto -1; queste meritano un commento. Le priorità 16-31 sono chiamate realtime, ma non lo sono: non vi sono garanzie e non si controllano scadenze, semplicemente, sono priorità più alte di 0-15. Ad ogni modo, le priorità che variano da 16 a 31 sono riservate al sistema, ed ai thread con priorità esplicitamente assegnata a tali valori dall'amministratore di sistema. Gli utenti ordinari non possono operare in questa fascia per ovvie ragioni: se un thread utente volesse girare ad una priorità più elevata, diciamo, del thread della tastiera o del mouse, ed entrasse in un loop, il thread del mouse o della tastiera non andrebbe mai in esecuzione, bloccando di fatto l'intero sistema.

I thread utente girano con priorità 0-15; assegnando le priorità dei processi e dei thread, un utente può dare la preferenza ad un certo thread. Il thread zero gira in background e consuma il tempo di CPU che nessun altro vuole; ha il compito di azzerare le pagine per il gestore della memoria. Se non vi è proprio nulla da fare, neanche azzerare pagine, viene mandato in esecuzione il thread inattivo, sebbene questo non sia un thread vero e proprio.

Nel corso del tempo, sono state fatte alcune rettifiche all'algoritmo di scheduling di base per migliorare le prestazioni del sistema; in certe condizioni specifiche, la priorità corrente di un thread utente può essere aumentata oltre la priorità di base (dal sistema operativo), ma mai sopra il valore 15, e poiché l'array in Figura 11.19 è basato sulla priorità corrente, il cambiamento di questa priorità influenza lo scheduling. Non si fa mai nessun adattamento per i thread a priorità 15 o superiore.

Vediamo adesso quando viene aumentata la priorità di un thread: in primo luogo quando si completa un'operazione di I/O che rilascia un thread in attesa, la priorità vie-

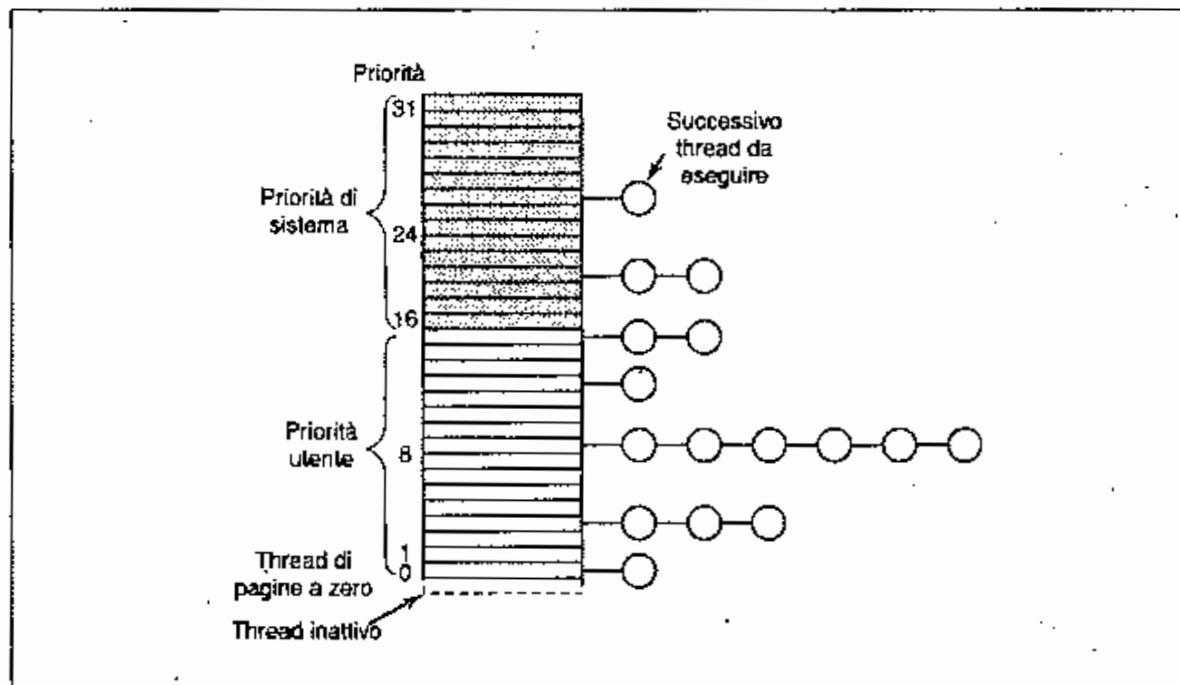


Figura 11.19 Windows 2000 supporta 32 priorità per i thread.

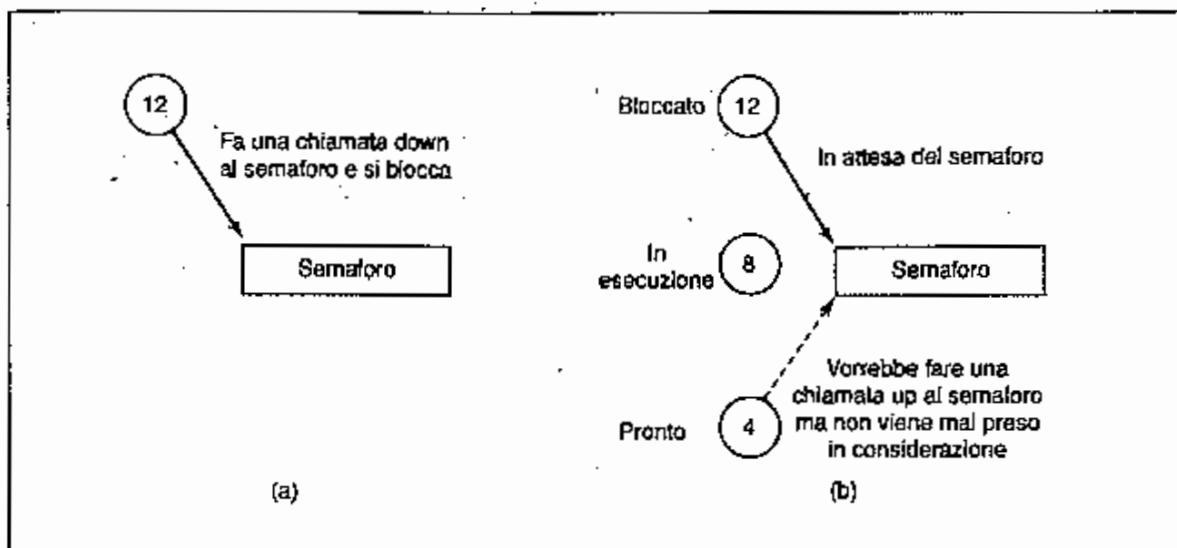


Figura 11.20 Un esempio di inversione di priorità.

ne spinta verso l'alto per dargli la possibilità di girare ancora subito, ed incominciare altre operazioni di I/O. L'idea qui è quella di mantenere i dispositivi di I/O occupati al massimo; l'ammontare delle spinte dipende dal dispositivo di I/O, tipicamente 1 per il disco, 2 per la linea seriale, 6 per la tastiera e 8 per la scheda sonora.

In secondo luogo, se un thread era in attesa su un semaforo, su un mutex o anche su un evento, quando viene rilasciato riceve una spinta di 2 unità se si trova nel processo in primo piano (il processo che controlla la finestra nella quale sta operando la tastiera) e di 1 unità altrimenti. Questa situazione tende ad elevare i processi interattivi al di sopra della folla a livello 8; infine, se un thread GUI si sveglia poiché è disponibile dell'input dalla finestra, esso riceve una spinta per la stessa ragione.

Queste spinte non sono per sempre; hanno effetto immediato, ma se un thread utilizza tutto il suo quanto, esso perde un punto e si muove in giù nella coda dell'array di priorità. Se utilizza ancora un quanto intero, scende ancora di un altro livello, e così fino al suo livello base, dove rimane fino a quando non viene spinto nuovamente. Chiaramente, se un thread vuole un buon servizio deve suonare molta musica.

Vi è un altro caso in cui il sistema gioca con le priorità: si immagini due thread che stanno lavorando insieme per un problema di tipo produttore-consutatore. Il lavoro del produttore è più duro, quindi esso ha priorità più alta, diciamo 12, in confronto al consumatore che ha priorità 4. Ad un certo punto, il produttore ha completato il buffer condiviso e si blocca su un semaforo, come illustrato in Figura 11.20(a).

Prima che il consumatore possa andare nuovamente in esecuzione, un thread non collegato a priorità 8 passa in stato pronto e incomincia a girare, come mostrato in Figura 11.20(b); finché questo thread è in grado di rimanere in esecuzione, ci rimarrà, poiché supera la priorità del consumatore, ed il produttore, con priorità più alta, rimane bloccato. In tali circostanze, il produttore non otterrà mai la possibilità di tornare in esecuzione fino a quando il thread con priorità 8 non avrà smesso.

Windows 2000 risolve questo problema con quello che potrebbe essere chiamato caritativamente un trucco sporchissimo: il sistema tiene traccia di quanto tempo è passato da quando il thread pronto è andato in esecuzione per l'ultima volta, e se questo lasso di tempo supera una certa soglia, il thread viene spostato a priorità 15 per 2 quanti. Questo potrebbe sbloccare il produttore, ma dopo due quanti, l'incremento viene brutalmente rimosso (piuttosto che decadere gradualmente). Probabilmente una soluzione migliore

poteva essere quella di penalizzare i thread che usano i loro quanti ripetutamente abbassando la loro priorità; dopo tutto, il problema non è stato causato dal thread affamato (starved) ma dal thread goloso (greedy). Questo problema è ben noto sotto il nome di **inversione di priorità**.

Un problema analogo avviene quando un thread con priorità 16 afferra un mutex e non ottiene per lungo tempo il permesso di andare in esecuzione, affamando thread più importanti di sistema, che stanno aspettando il mutex. Il sistema operativo può impedire ciò lasciando che un thread che ha bisogno di un mutex possa disabilitare lo scheduler finché è occupato; sui multiprocessori occorre usare uno spin lock.

Prima di abbandonare l'argomento scheduling, vale la pena di spendere un paio di parole sul quanto. In Windows 2000 Professional il valore di base è 20 ms; sui server uniprocessore è 120 ms; sui multiprocessori vengono utilizzati vari altri valori, in base alla frequenza del clock. Più piccolo è il quanto, più si favoriscono gli utenti interattivi, mentre il quanto grande diminuisce i cambi di contesto e ciò aumenta l'efficienza; questo è il significato dell'ultima colonna di Figura 11.2. Questi valori di base possono essere manualmente moltiplicati per 2, 4 o 6, se lo si desidera; si ricordi che la dimensione del quanto è stata scelta dieci anni fa, e non è più cambiata da allora, sebbene le macchine adesso siano oltre un ordine di grandezza più veloci. Tali numeri potrebbero essere diminuiti di un fattore di 5 o 10 senza danneggiare, e migliorando probabilmente, il tempo di risposta per i thread interattivi in un sistema fortemente carico.

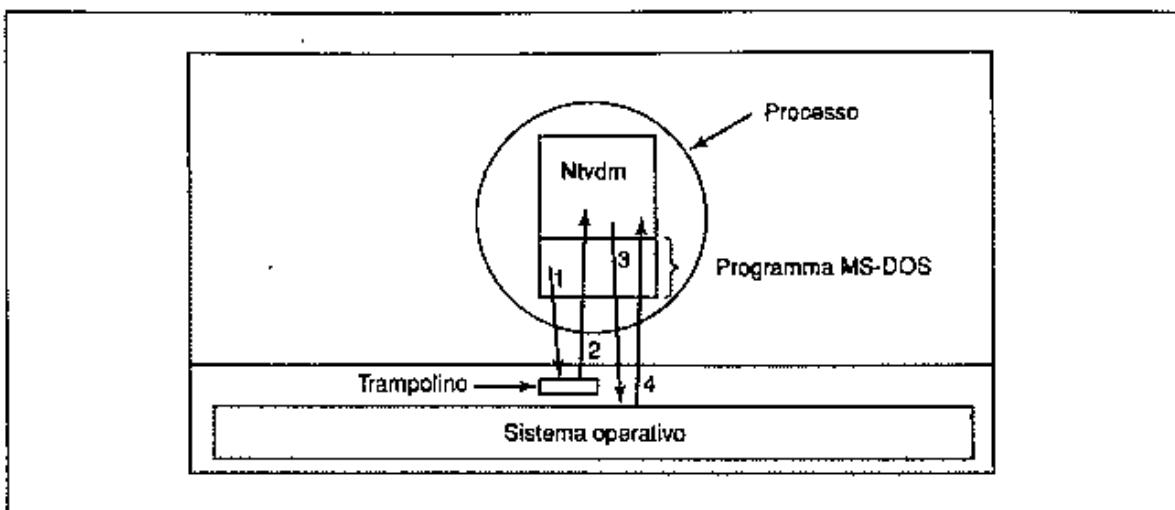
Un'ultima rettifica all'algoritmo di scheduling fa sì che quando una nuova finestra passa in primo piano, tutti i suoi thread prendano un quanto più lungo, aumentato di una quantità presa dal registro. Questo cambiamento dà ai thread più tempo di CPU, che di solito si traduce in un servizio migliore per la finestra che si è appena mossa in primo piano.

#### 11.4.4 L'emulazione di MS-DOS

Uno degli scopi del progetto di Windows 2000 era ereditato da NT: provare a far girare quanti più programmi MS-DOS possibile. Questa finalità era diversa dallo scopo di Windows 98: far girare tutti i vecchi programmi MS-DOS (ai quali noi aggiungiamo: non importa quanto male possano comportarsi).

Il modo in cui Windows 2000 si comporta rispetto ai vecchi programmi è di farli girare in un ambiente completamente protetto. Quando un programma MS-DOS inizia, si esegue un normale processo Win32, e si carica un programma emulatore MS-DOS, *ntvdm* (NT Virtual DOS Machine – Macchina Virtuale NT per DOS) che controllerà il programma MS-DOS eseguendo le chiamate di sistema. Poiché MS-DOS riconosce solo la memoria fino a 1MB sull'8088 e, con commutazione di banchi di memoria ed altri stratagemmi, fino a 16MB nel 286, è sicuro porre *ntvdm* in alto nello spazio degli indirizzi virtuale del processo, dove il programma non ha modo di scrivervi sopra; questa situazione è mostrata in Figura 11.21.

Quando il programma MS-DOS sta eseguendo istruzioni normali, esso può essere eseguito sull'hardware nudo poiché il Pentium ha come sottoinsieme tutte le istruzioni del 286 e dell'8088. La parte interessante si ha quando un programma MS-DOS vuole fare un'operazione di I/O o interagire con il sistema; un programma che si comporta in modo legale fa semplicemente una chiamata di sistema, ed in attesa di ciò, *ntvdm* istruisce Windows 2000 a riflettergli tutte le chiamate di sistema MS-DOS. In effetti, la chiamata di sistema salta semplicemente il sistema operativo e viene catturata dall'emulatore, come mostrato in Figura 11.21 nei passi 1 e 2; certe volte questa tecnica viene chiamata **trampolino**.



**Figura 11.21** Come i vecchi programmi MS-DOS vengono eseguiti sotto Windows 2000.

Una volta ottenuto il controllo, l'emulatore capisce che cosa vuole fare il programma ed emette le proprie chiamate Win32 per far fare il lavoro (passi 3 e 4 in Figura 11.21); se il programma si comporta bene e fa chiamate di sistema MS-DOS legali, questa tecnica funziona perfettamente. Il problema è che alcuni vecchi programmi MS-DOS evitavano il sistema operativo e scrivevano direttamente sulla RAM video, leggevano direttamente da tastiera e così via, cose che sono impossibili da fare in un ambiente protetto. Nella misura in cui il comportamento illegale causa una trap, vi è la speranza che l'emulatore possa scoprire che cosa vuole fare il programma ed emularlo, ma se non è in grado di capire cosa vuole il programma, lo uccide, semplicemente, perché il 100% dell'emulazione non è un requisito di Windows 2000.

## 11.4.5 Avviare Windows 2000

Prima che Windows 2000 possa partire, bisogna avviarlo; il processo di avvio crea dei processi iniziali che fanno partire il sistema, ed in questa sezione discuteremo brevemente di come lavora il processo di avvio per Windows 2000. Il processo di avvio hardware legge il primo settore del primo disco (il master boot record) e salta in esso, come descritto nella Sezione 5.4.2. Questo breve programma assembler legge la tabella delle partizioni per vedere quale partizione contiene il sistema operativo; quando la trova, legge il primo settore che può essere avviato, detto settore di avvio (boot sector), e salta ad esso. Il programma nel settore di avvio legge la directory radice della partizione, cercando il file *ntldr* (un altro reperto archeologico che mostra che Windows 2000 è veramente Windows NT); se lo trova, lo legge in memoria e lo esegue. *ntldr* carica Windows 2000. Per inciso, vi sono parecchie versioni del settore di avvio, a seconda di come la partizione è stata formattata, come FAT-16, FAT-32 o NTFS; quando Windows 2000 viene installato, si scrivono su disco le versioni corrette del master boot record e del settore di avvio.

*Ntldr* legge quindi il file *Boot.ini*, unica informazione di configurazione non presente nel registro; questo file elenca tutte le versioni di *hal.dll* e *ntoskrnl.exe* disponibili per l'avvio nella partizione, ed inoltre fornisce altri parametri, ad esempio quante CPU è possibile utilizzare e quanta RAM, se dare ai processi utente 2GB o 3GB, e quale velocità assegnare al clock in tempo reale. *Ntldr* quindi seleziona e carica i file *hal.dll* e *ntoskrnl.exe* e

pure *bootvid.dll*, il driver video per difetto per scrivere sul monitor durante il processo di avvio. Di seguito, *ntldr* legge nel registro per vedere quali altri driver sono necessari per completare l'avvio (ad esempio i driver della tastiera e del mouse, ma anche dozzine di altri driver per il controllo di vari chip sulla scheda madre); infine, legge tutti questi driver e passa il controllo a *ntoskrnl.exe*.

Una volta avviato, il sistema operativo compie alcune procedure di inizializzazione e chiama, quindi, le componenti dell'esecutivo per fare le varie inizializzazioni; per esempio, il gestore di oggetti prepara il suo spazio dei nomi per permettere ad altre componenti di chiamarlo per inserire i loro oggetti nel suddetto spazio. Molte componenti fanno delle operazioni che sono specifiche delle loro funzioni, come il gestore della memoria, che definisce le tabelle di pagine iniziali, o il gestore del plug-and-play, che recupera quali dispositivi di I/O sono presenti e carica i driver relativi. Si devono compiere dozzine di passi, durante i quali una barra di avanzamento sullo schermo aumenta di lunghezza via via che i passi vengono completati uno dopo l'altro. L'ultimo passo è la creazione del primo vero processo utente, il gestore di sessione (*session manager*), *smss.exe*; una volta creato ed in esecuzione, la fase di avvio è da considerarsi completata.

Il gestore di sessione è un processo nativo di Windows 2000; fa vere chiamate di sistema e non utilizza il sottosistema di ambiente Win32, che non è ancora in esecuzione: in effetti uno dei suoi primi doveri è inizializzarlo (*csrss.exe*). Esso legge anche i dati del registro dal disco, e viene a conoscenza di tutto quello che gli serve; tipicamente deve inserire molti oggetti nello spazio dei nomi del gestore di oggetti, creando ogni file di paginazione supplementare necessario, e apre DLL importanti per averle disponibili al momento opportuno. Dopo aver fatto la maggior parte del lavoro crea un demone di login, *winlogon.exe*.

| Processo               | Descrizione                                                                 |
|------------------------|-----------------------------------------------------------------------------|
| idle                   | Non un vero processo ma la casa del thread inattivo                         |
| system                 | Crea smss.exe e i file di paginazione; legge il registro e apre le DLL      |
| <b>smss.exe</b>        | Il primo vero processo; soprattutto inizializzazione; crea csrss e winlogon |
| csrss.exe              | Processo di sottosistema Win32                                              |
| winlogon.exe           | Demone di autenticazione                                                    |
| lsass.exe              | Gestore delle autenticazioni                                                |
| services.exe           | Guarda nel registro e inizia i servizi                                      |
| Printer server         | Permette ai job remoti di utilizzare la stampante                           |
| File server            | Serve le richieste per i file locali                                        |
| Telnet daemon          | Permette le login in remoto                                                 |
| Incoming email handler | Accetta e memorizza le e-mail in arrivo                                     |
| Incoming fax handler   | Accetta e stampa i fax in arrivo                                            |
| DNS resolver           | Nome del server DNS Internet                                                |
| Event logger           | Registra vari eventi di sistema                                             |
| Plug-and-play manager  | Controlla l'hardware per vedere che dispositivi ci sono                     |

**Figura 11.22** I processi che partono durante la fase di avvio del sistema. Quelli sopra la linea orizzontale in grassetto devono sempre iniziare. Quelli sotto sono esempi di servizi che potrebbero iniziare.

A questo punto, il sistema operativo è partito ed è in esecuzione; è il momento di permettere ai processi di servizio (demoni nello spazio utente) di partire, ed agli utenti di entrare nel sistema. *Winlogon.exe* in primo luogo crea il gestore di autenticazioni (*lsass.exe*), e quindi il processo genitore di tutti i servizi (*services.exe*), il quale guarda nel registro per scoprire quali processi demoni sono necessari nello spazio utente, e in quali file si trovano; quindi incomincia a crearli. Essi sono mostrati in Figura 11.7; il fatto che il disco sia pesantemente utilizzato dopo che il primo utente entra nel sistema (ma prima che faccia operazioni) non è una colpa o sbaglio dell'utente: il colpevole è *services.exe*, che crea tutti i servizi, e poi carica tutti i driver di dispositivi rimanenti che non sono stati ancora caricati. La gerarchia dei processi iniziale e qualche servizio tipico sono mostrati in Figura 11.22.

*Winlogon.exe* è anche responsabile di tutti gli accessi degli utenti; il vero e proprio dialogo di accesso è gestito da un programma separato nella DLL, *msgina.dll*, ciò per rendere possibile a terze parti di sostituire il login di base con l'identificazione mediante impronte del viso o altro, piuttosto che con login e password. Dopo che è avvenuto un login con successo, *winlogon.exe* recupera il profilo utente dal registro e da esso determina quale interprete di comandi (shell) eseguire. Molti utenti non se ne rendono conto, ma il desktop di base di Windows non è altro che il processo *explorer.exe* con alcune opzioni predefinite. Se lo desidera, un utente può selezionare un qualsiasi altro programma come interprete di comandi, come il prompt di comandi o anche Word, editando opportunamente il registro. Ad ogni modo, editare il registro non è una operazione per deboli di cuore; un errore può rendere il sistema inutilizzabile.

## 11.5 La gestione della memoria

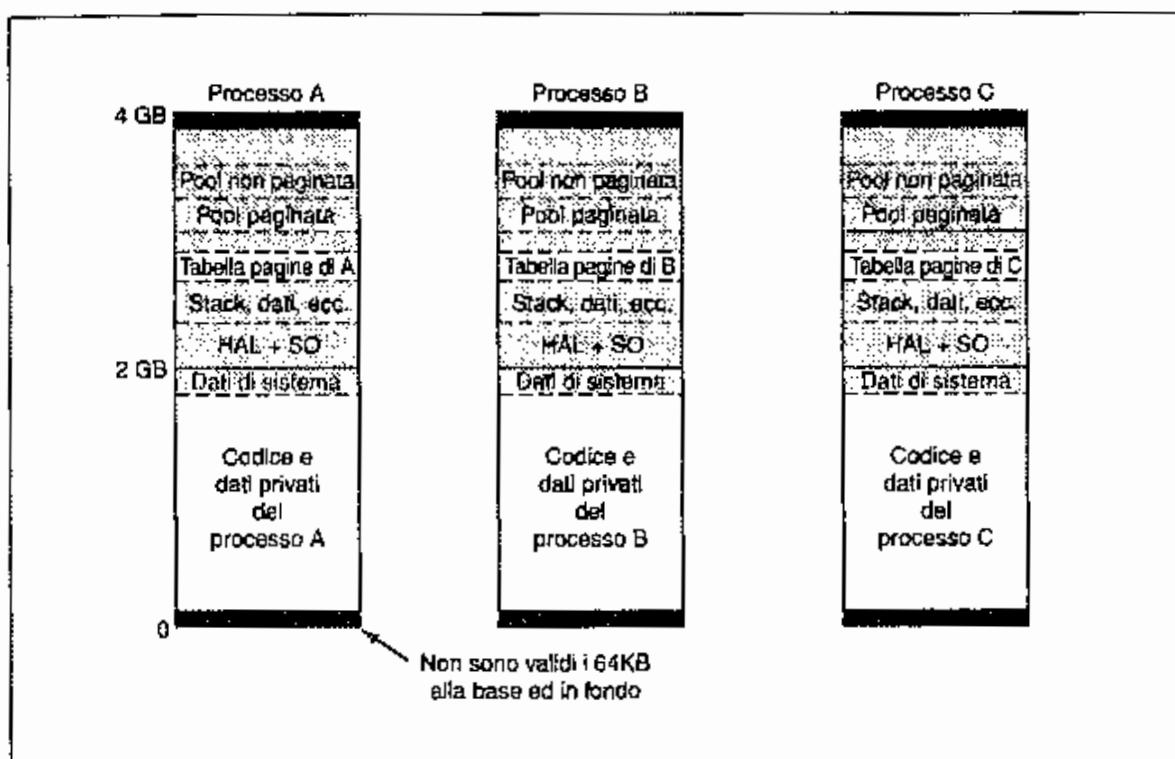
Windows 2000 dispone di un sistema di memoria virtuale estremamente sofisticato. Esso ha un certo numero di funzioni Win32 per utilizzarlo, e parte dell'esecutivo più sei thread di kernel sono dedicati a gestirlo. Nella sezione seguente guarderemo i concetti fondamentali, le chiamate API Win32, ed infine la sua implementazione.

### 11.5.1 Concetti di base

In Windows 2000, ogni processo utente ha il proprio spazio di indirizzi virtuali; gli indirizzi virtuali sono lunghi 32 bit, ed ogni processo ha uno spazio degli indirizzi virtuale di 4GB. I 2GB più bassi meno 256MB sono disponibili per il codice del processo ed i dati; i 2GB superiori mappano la memoria del kernel in modo protetto. Lo spazio degli indirizzi virtuale è paginato a richiesta, con dimensione fissata (4KB sul Pentium).

La disposizione dello spazio degli indirizzi virtuale per tre processi utente è mostrata, in modo abbastanza semplificato, in Figura 11.23; normalmente i 64KB alla base ed in testa ad ogni spazio degli indirizzi virtuale di un processo non sono mappati. Questa scelta è stata fatta intenzionalmente per permettere di cogliere gli errori di programmazione: i puntatori non validi sono solitamente 0 o -1, e tentativi di usarli in Windows 2000 causerebbero immediatamente una trap software, invece di una lettura di dati casuali o, peggio ancora, la scrittura in una locazione sbagliata. Ad ogni modo, quando un vecchio programma MS-DOS sta girando con l'emulatore, questi KB possono essere mappati.

Dopo i 64KB incomincia il codice privato dell'utente ed i dati, che si estende fino ad un massimo di 2GB. L'ultimo pezzo dei 2GB contiene alcuni contatori di sistema e timer



**Figura 11.23** La disposizione dello spazio degli indirizzi virtuale per tre processi utente. Le aree bianche sono private per i processi, le aree ombreggiate sono condivise da tutti i processi.

condivisi in sola lettura tra tutti gli utenti; rendendoli visibili qui, i processi possono accedervi senza un sovraccarico di chiamate di sistema.

I 2GB più in alto contengono il sistema operativo, compresi il codice, i dati, e le pool paginate e non (usate per gli oggetti, eccetera); essi sono condivisi da tutti i processi utente, fatta eccezione per le tabelle di pagina, che sono tabelle di pagina proprie del singolo processo. I 2GB di memoria più in alto non sono scrivibili ed in genere neanche leggibili dai processi in modo utente. La ragione per cui sono messi qui è che quando un thread fa una chiamata di sistema si ha una trap in modalità kernel, il quale gira nello stesso thread. Rendendo visibile l'intero sistema operativo e tutte le sue strutture dati (come l'intero processo utente) all'interno dello spazio degli indirizzi del thread, quando questo entra in modo kernel, non è necessario cambiare la mappa di memoria o ripulire la cache; basta passare dallo stack del thread allo stack del kernel nel thread. Il compromesso qui è uno spazio degli indirizzi privato per ogni processo più piccolo, in cambio di chiamate di sistema più veloci. I server per grosse basi dati risentono di questo limite, il che è la ragione per la quale sono opzionalmente disponibili 3GB di spazio utente nella versione Server Avanzato e Server Datacenter.

Ogni pagina virtuale si può trovare in uno dei tre stati seguenti: libera, riservata e impegnata. Una pagina libera (*free page*) non è attualmente in uso, ed un riferimento ad essa causa un fault di pagina. Quando un processo inizia, tutte le sue pagine sono libere fino a quando il programma ed i dati iniziali non vengono mappati nel suo spazio degli indirizzi; dopo che il codice o i dati sono stati mappati in una pagina, la pagina viene detta impegnata (*committed*). Un riferimento ad una pagina impegnata viene mappato usando l'hardware per la memoria virtuale, ed ha successo se la pagina si trova in memoria principale, mentre se non si trova nella memoria principale, avviene un

fault di pagina ed il sistema operativo trova e prende la pagina in questione dal disco.

Una pagina virtuale può essere anche **riservata (reserved)**, il che vuol dire che non può essere mappata fino a quando la prenotazione non viene esplicitamente rimossa; per esempio, quando si crea un nuovo thread, si riserva 1MB di spazio di stack nello spazio degli indirizzi virtuali del processo, ma risulta impegnata solo una pagina. Questa tecnica permette che lo stack possa eventualmente crescere fino ad 1MB senza rischiare che qualche altro thread allochi lo spazio contiguo necessario dallo spazio degli indirizzi virtuali, portandoglielo via. Oltre a poter essere nei tre stati, libera, riservata e impegnata, una pagina possiede anche molti altri attributi come il fatto di essere leggibile, scrivibile ed eseguibile.

Un compromesso interessante avviene assegnando spazi di memorizzazione su disco alle pagine impegnate; una strategia semplice potrebbe essere quella di assegnare lo spazio nel file di paginazione al momento in cui la pagina è impegnata, in questo modo conosciamo sempre il posto dove scrivere per esteso le pagine ritirate dalla memoria fisica. Il lato negativo di questa strategia è che il file di paginazione potrebbe diventare grande quanto la somma della memoria virtuale di tutti i processi; in un sistema con molta memoria che raramente la finisce, e che vada a paginare raramente, questo approccio sprecherebbe spazio disco.

Per evitare di sprecare spazio, le pagine impegnate di Windows 2000 che non necessitano di finire su disco, ad esempio pagine di stack, non hanno uno spazio su disco fino al momento in cui non vengono paginate. Questo approccio rende il sistema più complesso, perché le mappe dei file di paginazione varano recuperate durante un fault di pagina, e recuperandole si potrebbero causare uno o più fault di pagina supplementari all'interno del gestore dei fault di pagina. D'altro canto, non si alloca spazio per le pagine che non sono mai paginate fuori.

Compromessi come questo (complessità del sistema contro migliori prestazioni o più funzionalità), tendono ad essere risolti in favore di queste ultime, perché il valore di migliori prestazioni o più funzionalità è chiaro, ma la diminuzione di complessità (una maggiore difficoltà di manutenzione e più blocchi del sistema ogni anno) è difficile da quantificare. Le pagine riservate e libere non hanno mai pagine ombra su disco, e i riferimenti ad esse causano sempre fault di pagina.

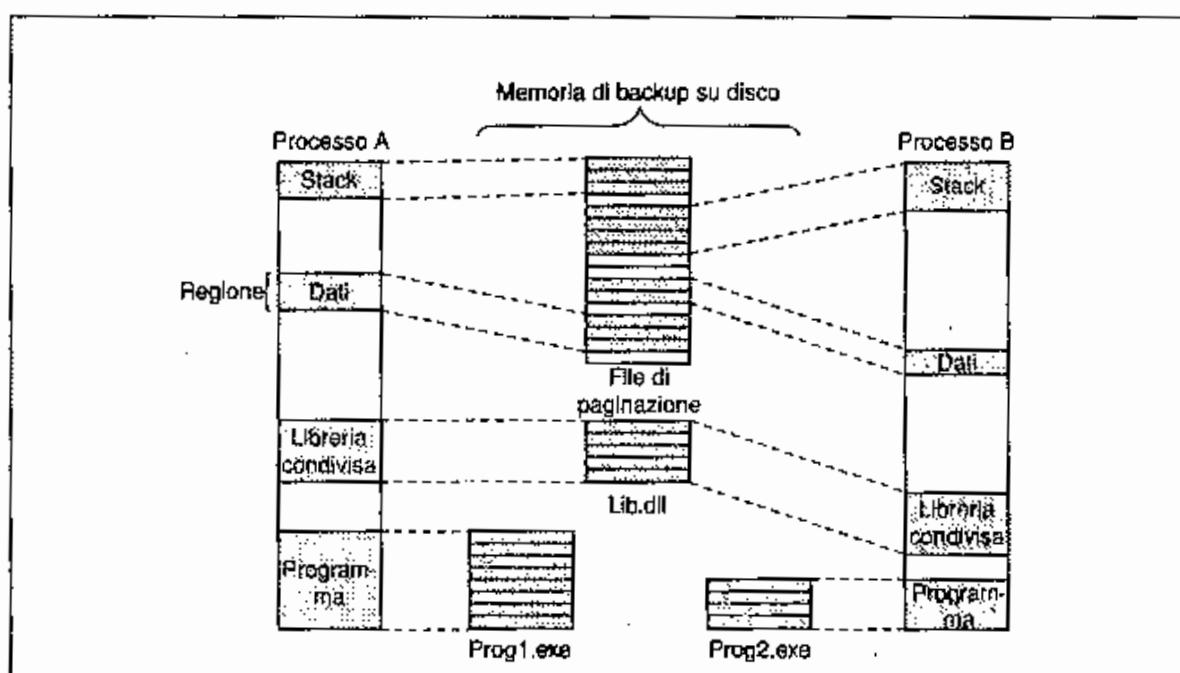
Le pagine ombra del disco sono gestite in uno o più file di paginazione, fino a 16, e possono essere distribuiti su 16 dischi separati per una maggiore banda di trasferimento in I/O. Ogni file ha una dimensione iniziale e una dimensione massima, che può raggiungere in caso di necessità; ma questi file possono essere creati con la dimensione massima al momento dell'installazione del sistema, in modo da ridurre la possibilità che siano fortemente frammentati, oppure ne possono essere creati di nuovi successivamente utilizzando il pannello di controllo. Il sistema operativo tiene traccia della mappa delle pagine virtuali all'interno dei file di paginazione. Per i programmi (file solo eseguibili) il file binario eseguibile (cioè file .exe o .dll) contiene le pagine ombra; per le pagine dati, si usano i file di paginazione.

Windows 2000, come molte versioni di UNIX, permette che i file siano mappati direttamente in regioni degli spazi virtuali di indirizzi (cioè serie di pagine consecutive). Una volta che un file è stato mappato su uno spazio degli indirizzi, può essere letto e scritto usando i normali riferimenti alla memoria. I file mappati in memoria sono implementati come le pagine impegnate, solo che le loro pagine ombra sono in un file utente invece che in un file di paginazione. Come risultato di ciò, mentre un file viene mappato, la versione in memoria potrebbe non essere identica alla versione su disco (a causa di scritture recenti nello spazio virtuale di indirizzi), ma quando il file non viene più mappato, o il file viene esplicitamente ricopiatato su disco, la versione del disco diventa aggiornata.

Windows 2000 ammette esplicitamente che due o più processi si mappino sulla stessa parte dello stesso file nello stesso momento, se possibile utilizzando indirizzi virtuali diversi, come mostrato in Figura 11.24. Leggendo e scrivendo le parole di memoria, i processi possono ora comunicare in modo molto efficiente, e possono passarsi avanti ed indietro i dati ad alta velocità, poiché non avviene nessuna operazione di copiatura; processi diversi possono avere diversi permessi di accesso. Poiché tutti i processi che stanno utilizzando un file mappato condividono le stesse pagine, i cambiamenti fatti da uno di loro diventano immediatamente visibili a tutti gli altri, anche se il file su disco non è stato ancora aggiornato. Occorre fare attenzione che, se un altro processo apre il file in lettura normale, esso veda le pagine correnti in RAM e non le pagine vecchie sul disco.

Occorre notare che vi è un problema nel caso in cui due programmi condividano un file DLL, ed uno dei due cambi i dati statici del file; se non si fa nessuna azione particolare, l'altro programma vedrà i dati cambiati, il che, probabilmente, non è desiderabile. Il problema viene risolto mappando tutte le pagine come in sola lettura, ma annotando in segreto che in realtà qualcuna potrebbe essere scrivibile; quando una scrittura viene fatta in una pagina che è mappata in sola lettura ma è in realtà scrivibile, si fa una copia privata della pagina e la si mappa. Ora questa può essere scritta in modo sicuro senza influenzare altri utenti né la copia originale su disco. Questa tecnica è chiamata *copy-on-write* (*copia su scrittura*).

È anche importante notare che se il codice del programma è mappato in due spazi di indirizzi ad indirizzi differenti, si verifica un problema di indirizzamento. Cosa succede se la prima istruzione è JMP 300? Se il primo processo mappa il programma all'indirizzo 65536, il codice può facilmente essere corretto diventando JMP 65836. Ma cosa succede se un secondo processo adesso lo mappa all'indirizzo 131072? L'istruzione JMP 65836 andrà all'indirizzo 65836 invece di 131372 ed il programma andrà in errore. La soluzione è utilizzare nel codice che deve essere condiviso solo salti relativi, non indi-



**Figura 11.24** Le regioni mappate con le loro pagine ombra sul disco. Il file Lib.dll viene mappato in due spazi di indirizzi nello stesso momento.

rizzi virtuali assoluti. Fortunatamente, molte macchine hanno istruzioni che utilizzano salti relativi oltre ad istruzioni che utilizzano indirizzi assoluti. I compilatori possono utilizzare le istruzioni di salto relativo, ma devono sapere in anticipo se utilizzare quelle in modo assoluto o relativo; il salto relativo non è sempre utilizzato poiché il codice prodotto è di solito meno efficiente. In linea generale, un flag del compilatore indica quale tipo di salto utilizzare; la tecnica che rende possibile porre pezzi di codice a qualsiasi indirizzo virtuale senza riallocazione è detta codice indipendente dalla posizione (*position independent code*).

Anni fa, quando gli spazi di indirizzi virtuali da 16-bit (o 20-bit) erano uno standard, ma le macchine avevano megabyte di memoria fisica, vennero pensati tutti i tipi di trucchi per permettere ai programmi di utilizzare più memoria fisica di quella disponibile nello spazio di indirizzi. Spesso questi trucchi andavano sotto il nome di **commutazione di banchi** (*bank switching*), nei quali il programma poteva sostituire qualche blocco di memoria al di fuori del limite dei 16-bit (20-bit) con un blocco della sua stessa memoria. Quando vennero introdotte le macchine a 32-bit, si iniziò a pensare che non ci sarebbero mai più stati dei problemi con lo spazio degli indirizzi: fu un errore. Il problema si ripresenta, perché i programmi di grosse dimensioni necessitano che Windows 2000 allochi loro più di 2GB o 3GB di spazio di indirizzamento utente, così la commutazione di banchi è ritornata, ma adesso è chiamata **Estensione dell'indirizzamento a finestre** (*address windowing extension*). Questa facilitazione permette ai programmi di mappare pezzi mischiati di memoria dentro e fuori lo spazio di indirizzamento dell'utente (e specialmente sopra il temuto confine dei 4GB). Dato che è utilizzata solo su server con più di 2GB di memoria fisica, rinviiamo la discussione alla prossima edizione di questo libro (a quel tempo, anche i computer da tavolo di base assorberanno le restrizioni dei 32-bit).

### 11.5.2 Le chiamate di sistema per la gestione della memoria

L'API Win32 contiene un certo numero di funzioni che permettono ad un processo di gestire in modo esplicito la sua memoria virtuale; le più importanti tra queste funzioni sono elencate in Figura 11.25. Ognuna di loro opera su una regione composta o di una singola pagina o di una sequenza di due o più pagine consecutive nello spazio di indirizzi virtuale.

Le prime quattro funzioni API sono utilizzate per allocare, liberare, proteggere e richiedere regioni dello spazio di indirizzi virtuali; le regioni allocate incominciano sempre sul confine dei 64 KB per minimizzare i problemi di migrazione o porting ad architetture future che avranno pagine più grandi di quelle correnti (al massimo 64 KB). La quantità di spazio di indirizzi effettivamente allocata può essere meno di 64 KB, ma deve essere un multiplo della dimensione della pagina. Le successive due danno al processo la possibilità di legare (e slegare) in modo hardware le pagine in memoria, in modo da non paginarle fuori; per esempio un programma in tempo reale potrebbe aver bisogno di questa funzionalità. Il sistema operativo pone però un limite per impedire che i processi diventino troppo golosi; di fatto, le pagine possono essere rimosse dalla memoria, ma solamente se l'intero processo viene portato su disco. Quando viene riportato indietro, tutte le pagine bloccate vengono ricaricate prima che i thread possano ricominciare la propria esecuzione. Sebbene non mostrato in Figura 11.25, Windows 2000 possiede anche funzioni API che permettono ad un processo di accedere alla memoria virtuale di un processo differente sul quale ha il controllo (cioè per il quale, possiede un gestore).

Le ultime quattro funzioni API nell'elenco sono dedicate alla gestione dei file mappa-

| Funzione Win32 API | Descrizione                                                            |
|--------------------|------------------------------------------------------------------------|
| VirtualAlloc       | Riserva o impegna una regione                                          |
| VirtualFree        | Rilascia o disimpegna una regione                                      |
| VirtualProtect     | Cambia i diritti di lettura/scrittura/esecuzione di una regione        |
| VirtualQuery       | Interroga per sapere lo stato di una regione                           |
| VirtualLock        | Rende una regione residente in memoria (cioè disattiva la paginazione) |
| VirtualUnlock      | Rende una regione paginabile al solito modo                            |
| CreateFileMapping  | Crea un oggetto per file mappabile e (optionalmente) gli dà un nome    |
| MapViewOfFile      | Mappa un file (o una parte) nello spazio degli indirizzi               |
| UnmapViewOfFile    | Rimuove un file mappato nello spazio degli indirizzi                   |
| OpenFileMapping    | Apri un oggetto per la mappa di file creato in precedenza              |

**Figura 11.25** Le principali funzioni API Win32 per la gestione della memoria virtuale in Windows 2000.

ti in memoria. Per mappare un file si deve prima creare un oggetto di mappa di file (vedi Figura 11.10), usando `CreateFileMapping`; questa funzione restituisce un gestore all'oggetto di mappa di file e, se richiesto, gli dà un nome nel file system, in modo che un altro processo lo possa usare. Le due funzioni successive rispettivamente aggiungono e rimuovono un file mappato nello spazio di indirizzamento. L'ultima funzione può essere utilizzata da un processo per mappare un file già mappato da un altro processo; in questo modo, due o più processi possono condividere delle regioni dei loro spazi di indirizzi. Questa tecnica permette di scrivere in regioni (limitate) della memoria di un altro processo.

### 11.5.3 Implementazione della gestione della memoria

Windows 2000 supporta un singolo spazio degli indirizzi lineare di 4 GB, paginato su richiesta per ogni processo; la segmentazione non è supportata in nessuna forma. Teoricamente le dimensioni delle pagine possono essere una qualsiasi potenza di 2 fino a 64 KB, ma sul Pentium queste dimensioni sono fissate a 4 KB. Sull'Itanium esse possono essere 8 o 16 KB; in più, il sistema operativo stesso può usare pagine a 4 MB per ridurre lo spazio consumato dalla tabella delle pagine.

A differenza dello scheduler, che seleziona thread singoli da eseguire, e non si preoccupa dei processi, il gestore della memoria ha interamente a che fare con i processi, e non si preoccupa dei thread: dopotutto, sono i processi (e non i thread) che possiedono lo spazio di indirizzi, e cioè l'oggetto con cui ha a che fare il gestore della memoria. Quando si alloca una regione dello spazio di indirizzi virtuale, (ce ne sono quattro per il processo A in Figura 11.24), il gestore della memoria crea per essa un VAD (Virtual Address Descriptor - Descrittore Virtuale di Indirizzi) elencando l'intervallo di indirizzi mappati, il file di salvataggio su disco della memoria virtuale, la lunghezza della mappa, ed infine il codice di protezione. Quando si tocca la prima pagina, si crea la directory delle tabelle di pagina, e si inserisce nel VAD un puntatore ad essa; in effetti, uno spazio di indirizzi è completamente definito dalla lista dei suoi VAD. Questo schema supporta gli spazi di indirizzi sparsi, visto che non si utilizzano risorse per le aree non utilizzate tra regioni mappate.

## Gestione di un fault di pagina

Windows 2000 non utilizza nessuna forma di pre-paginazione: quando inizia un processo, non è in memoria nessuna delle sue pagine, e tutte vengono caricate dinamicamente solo quando avviene un fault di pagina. Per ogni fault di pagina avviene un'interruzione software (trap) al Kernel (nel senso di Figura 11.7); il kernel costruisce un descrittore indipendente dalla macchina che esprime cosa è accaduto, e passa tale descrittore al gestore della memoria che è parte dell'esecutivo. Il gestore della memoria quindi controlla la sua validità; se una pagina fallita cade all'interno di una regione riservata o impegnata, il gestore guarda l'indirizzo corrispondente nella lista dei VAD, trova (o crea) la tabella della pagina, e vi cerca il dato corrispondente.

I campi della tabella di pagina sono differenti per architetture diverse, per il Pentium, la struttura di una pagina mappata è mostrata in Figura 11.26. Anche le pagine non mappate hanno una struttura, ma il loro formato è leggermente diverso; per esempio, se si deve azzerare una pagina non mappata prima di poterla utilizzare, lo si annota nella tabella della pagina.

I bit più importanti nella struttura della tabella di pagina, per gli scopi dell'algoritmo di paginazione, sono i bit A e D: essi sono impostati dall'hardware e tengono traccia rispettivamente se, dall'ultima volta che sono stati puliti, la pagina è stata riferita, o vi è stato scritto sopra.

I fault di pagina si suddividono in 5 categorie:

1. La pagina riferita non è impegnata.
2. È avvenuta una violazione di protezione.
3. Una pagina condivisa è stata scritta.
4. Lo stack ha bisogno di aumentare la sua dimensione.
5. Una pagina riferita è impegnata, ma non è al momento mappata.

Il primo e secondo caso sono errori fatali, da cui il processo che ha fallito non ha possibilità di recupero; il terzo caso ha gli stessi sintomi del secondo (un tentativo di scrivere sulla pagina in sola lettura), ma il comportamento è diverso. La soluzione è copiare la pagina in un nuovo frame di pagina fisico e mapparla in lettura e scrittura. Questa è la tecnica copy-on-write (se una pagina condivisa è marcata come scrivibile in tutti i processi che la stanno utilizzando, non può essere copy-on-write e non si verifica fault di pagina quando viene scritta). Il quarto caso richiede di allocare e mappare un nuovo frame di pagina; ad ogni modo, le regole di sicurezza richiedono che la pagina contenga solo

| Bit             | 20 | 3              | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1   |
|-----------------|----|----------------|---|---|---|---|---|---|---|-----|
| Frame di pagina |    | Non utilizzato | G | L | D | A | C | W | U | W/V |
|                 |    |                |   |   |   |   |   |   |   |     |

G: la pagina è globale a tutti i processi  
 L: pagina grande (4MB)  
 D: la pagina è sporca  
 A: ci sono stati accessi alla pagina  
 C: caching attivato/disattivato  
 W: scrivibile (senza caching)  
 U: la pagina è accessibile in modo utente  
 W: pagina scrivibile  
 V: campo dati per la tabella delle pagine valido

Figura 11.26 Un elemento della tabella di pagina per una pagina mappata su un Pentium.

zeri, per impedire che un processo osservi una pagina del proprietario precedente, quindi si deve trovare una pagina con tutti zeri e, se questa non fosse disponibile, si deve allocare ed azzerare un nuovo frame di pagina. Infine, il quinto caso è un normale fault di pagina, la pagina viene individuata e mappata.

Il meccanismo vero e proprio per prendere e mappare le pagine è standard, quindi non ne discuteremo; l'unica funzionalità degna di nota è che Windows 2000 non legge dal disco pagine isolate, bensì legge serie di pagine consecutive, di solito 1-8 pagine, nel tentativo di minimizzare il numero di trasferimenti dal disco. La dimensione di questa serie è più grande per pagine di codice piuttosto che per pagine di dati.

## L'algoritmo di sostituzione di pagine

La sostituzione di una pagina avviene nel modo che segue: il sistema cerca in ogni modo di mantenere un numero consistente di pagine libere in memoria, così, quando avviene un fault di pagina, si può richiedere immediatamente la pagina libera, senza la necessità di scrivere, prima, qualche altra pagina su disco. In conseguenza di questa strategia, si è in grado di soddisfare la maggior parte dei fault di pagina con una unica operazione disco (leggendo la pagina dal disco), piuttosto che due (scrivendo una pagina sporca su disco e poi leggendo la pagina necessaria).

Ovviamente, le pagine presenti nella lista libera provengono da qualche parte, quindi il vero algoritmo di sostituzione consiste nel recuperare le pagine dai processi e nel metterle nella lista libera (in effetti, vi sono quattro liste libere, ma per il momento è più semplice pensare ad una sola; entreremo in dettaglio più avanti). Vediamo ora come Windows 2000 libera le pagine: innanzitutto, l'intero sistema di paginazione opera usando pesantemente il concetto di insieme di lavoro. Ogni processo (*non* ogni thread) ha un insieme di lavoro (*working set*), formato da pagine mappate in memoria e che quindi possono essere riferite senza fault di pagina; ovviamente, la dimensione e la composizione dell'insieme di lavoro fluttua quando i thread del processo sono in esecuzione.

Ogni insieme di lavoro di un processo è definito da due parametri: la dimensione minima e quella massima. Non rappresentano limiti stretti, ovvero un processo può avere poche pagine in memoria - meno della dimensione minima - o (in certe circostanze) avere più pagine di quante consentite dalla dimensione massima. Ogni processo incomincia con gli stessi valori di massima e minima dimensione, che possono però variare nel tempo: per default il minimo è tra 20 e 50 ed il massimo tra 45 e 345 (in relazione alla quantità totale di RAM). Ad ogni modo, l'amministratore di sistema può cambiare questi valori in ogni momento.

Se avviene un fault di pagina, e l'insieme di lavoro è più piccolo del minimo, la pagina viene aggiunta, mentre se avviene un fault di pagina e l'insieme di lavoro è più grande del massimo, una pagina viene tolta dall'insieme di lavoro (ma non dalla memoria) per fare posto alla nuova pagina. Questo vuol dire che Windows 2000 utilizza un algoritmo locale, in modo da impedire che un processo possa danneggiarne altri invadendo la memoria. Ad ogni modo, il sistema cerca entro certi limiti di autoregolarsi; per esempio, se un processo sta chiedendo pagine disperatamente (e gli altri no), il sistema può aumentare la dimensione del suo insieme di lavoro, quindi nel tempo l'algoritmo opera sia localmente sia globalmente. Vi è, comunque, un limite assoluto sulla dimensione dell'insieme di lavoro: anche con un solo processo in esecuzione, essa non può superare le 512 pagine, in modo che rimanga un po' di spazio per i nuovi processi.

E fino a qui tutto bene, ma la storia non è ancora finita: una volta al secondo, un thread demone dedicato, il gestore di insiemi di bilanciamento, controlla se vi sono abbastanza pagine libere, e se non ve ne sono, attiva il thread gestore degli insiemi di lavoro per

esaminare gli insiemi di lavoro e recuperare più pagine. Tale gestore determina l'ordine con cui esaminare i processi: i processi grandi, rimasti inattivi per molto tempo, vengono considerati prima dei processi piccoli ed i processi in primo piano vengono considerati per ultimi.

Successivamente, il gestore degli insiemi di lavoro incomincia ad ispezionare i processi nell'ordine scelto. Se l'insieme di lavoro del processo corrente è più piccolo del suo minimo, o dall'ultima ispezione sono avvenuti un certo numero di fault di pagina, questo viene lasciato stare; altrimenti, vengono rimosse una o più pagine, in funzione della dimensione totale della RAM, di quanta memoria manca, dell'esito del confronto fra la dimensione corrente dell'insieme ed il massimo e minimo del processo, ed altri parametri. A turno, tutte le pagine vengono esaminate.

Su un monoprocesso, se il bit di riferimento alla pagina è a 0, il contatore associato alla pagina viene incrementato; se tale bit di riferimento è a 1, il contatore viene azzerato. Dopo il controllo, le pagine con contatore più grande vengono rimosse dall'insieme di lavoro ed il thread continua ad analizzare i processi fino a quando non sono state recuperate abbastanza pagine, quindi si ferma. Se invece dopo un passaggio completo su tutti i processi non si sono recuperate abbastanza pagine, ne fa un altro, esaminando in modo più aggressivo, e riducendo anche gli insiemi di lavoro ad una dimensione minore del loro minimo, se necessario.

In un multiprocessore, non è significativo guardare al bit di riferimento perché la CPU corrente potrebbe non aver toccato la pagina di recente, mentre altre CPU potrebbero averlo fatto; inoltre, esaminare tutti i bit di riferimento per tutte le altre CPU risulterebbe troppo dispendioso. Di conseguenza, il bit di riferimento viene trascurato e viene scelta, per essere rimossa, la pagina più vecchia.

Persino il sistema operativo, per la sostituzione di pagine, è pensato come un processo che possiede pagine e insieme di lavoro, ed è esaminato come gli altri. Ad ogni modo, parte del codice e il pool non paginato sono bloccati in memoria e non possono essere paginati neanche in circostanze speciali.

## La gestione della memoria fisica

Come detto in precedenza, esistono in effetti quattro liste libere: vediamo a cosa servono. Ogni pagina in memoria si trova in uno (o più) insiemi di lavoro o in una sola di queste quattro liste libere, come mostrato in Fig.11.27. Le liste standby (pulita) e modificata (sporca) contengono tutte quelle pagine che sono state recentemente eliminate da un insieme di lavoro, e sono residenti ancora in memoria ed associate ad un processo che le aveva usate in precedenza. La differenza fra queste due liste è che le pagine pulite hanno una copia valida su disco e quindi possono essere rilasciate immediatamente, mentre le pagine sporche non hanno una copia aggiornata sul disco. La lista libera contiene pagine pulite che non sono più associate a nessun processo; le pagine sulla lista delle pagine azzerate non sono associate ad alcun processo e sono tutte riempite da zeri. Una quinta lista contiene tutte le pagine fisiche di RAM che presentano difetti, per garantire che non siano utilizzate da nessuno.

Il gestore degli insiemi di lavoro ed altri thread demoni del kernel spostano le pagine tra i vari insiemi di lavoro e le varie liste. Esaminiamo le transizioni: quando il gestore degli insiemi di lavoro rimuove una pagina da un insieme di lavoro, la pagina si sposta alla base della lista standby o modificata, a seconda del suo stato di pulizia. Queste transizioni viene mostrata come (1). Le pagine su entrambe le liste appaiono ancora come pagine valide, tanto che se avviene un fault di pagina ed una di queste pagine diviene necessaria, la si rimuove dalla lista riportandola nell'insieme di lavoro senza alcun I/O

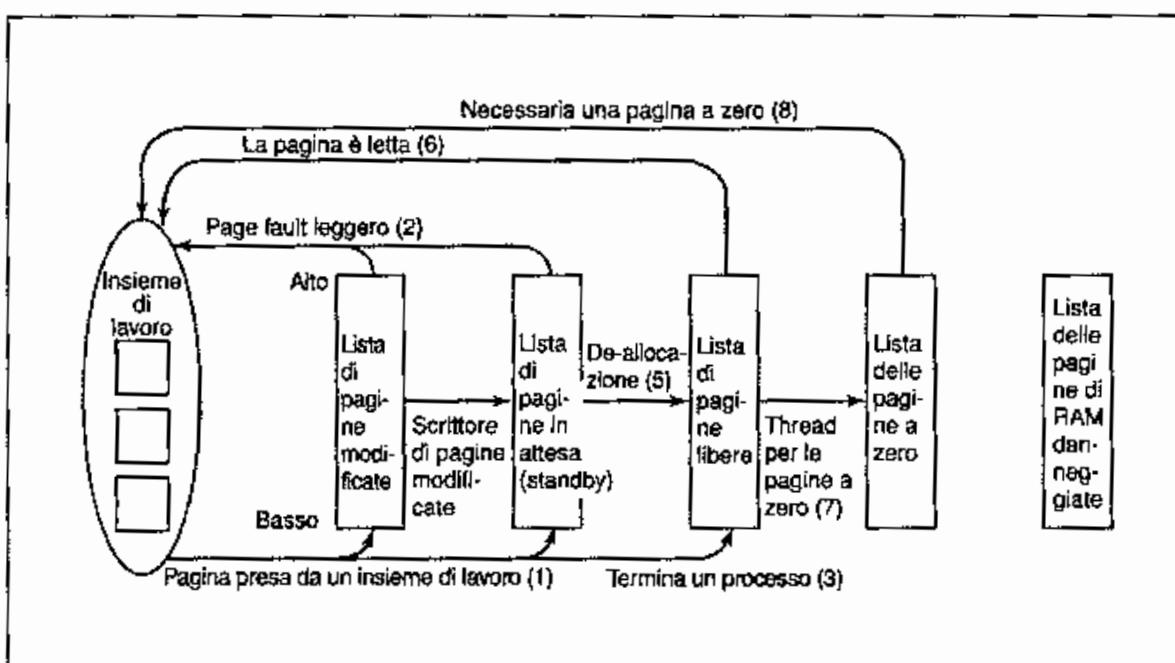


Figura 11.27 Le varie liste di pagine e le transizioni tra di loro.

su disco (2). Quando un processo termina, le sue pagine non condivise non potranno più essere riportate, quindi si spostano nella lista libera (3); queste pagine non sono più associate a nessun processo.

Altre transizioni vengono causate dai thread demoni: ogni 4 secondi il **thread swapper** entra in esecuzione e cerca i processi nei quali tutti thread sono rimasti inattivi per un certo numero di secondi; se ne trova qualcuno, gli stack del kernel sono smarcati e le loro pagine vengono spostate alle liste standby o modificata, come mostrato sempre in (1).

Due altri thread demoni, lo scrittore di pagine mappate (**mapped page writer**) e lo scrittore di pagine modificate (**modified page writer**), si svegliano periodicamente per vedere se vi sono abbastanza pagine pulite; se non ve ne sono, prendono le pagine dalla testa della lista modificata, le riscrivono su disco e, in seguito, le spostano nella lista standby (4). Il primo gestore scrive sui file inappati, il secondo sui file paginati; il risultato di queste scritture è di trasformare pagine sporche in pagine pulite.

La ragione per la quale esistono questi due thread è che un file mappato potrebbe aumentare di dimensioni in seguito ad una scrittura, e questa crescita potrebbe richiedere un accesso a strutture dati su disco per allocare blocchi di disco libero; se non vi è posto in memoria per portarvi tali strutture, quando si deve scrivere una pagina, potrebbe avvenire un deadlock. Il secondo thread può risolvere il problema scrivendo le pagine in un file di paginazione, che non cresce mai. Nessuno ha mai detto che Windows 2000 è semplice.

Le altre transizioni presenti in Figura 11.27 sono spiegate qui di seguito: se un processo elimina la mappa di una pagina, la pagina non è più associata al processo e può andare nella lista libera (5), eccetto nei casi in cui è condivisa. Quando un fault di pagina richiede un frame di pagina per mettervi la pagina che occorre leggere, il frame di pagina viene preso dalla lista libera (6), se possibile; non importa se la pagina contiene ancora informazioni confidenziali perché sta per essere riscritta in modo totale. La situazione è diversa, invece, quando lo stack cresce.

In quel caso, è necessario un frame di pagina vuoto, e le regole di sicurezza richiedo-

no che la pagina contenga tutti zeri; per questa ragione, un altro thread demone del kernel, il thread di azzeramento pagine (zero page thread), va in esecuzione alla priorità più bassa (vedi Figura 11.19), azzerando le pagine che sono nella lista libera, e mettendole nella lista di pagine azzerate (7). Se la CPU è inattiva e vi sono pagine libere, esse dovrebbero essere azzerate poiché una pagina azzerata è, potenzialmente, più utile di una pagina libera.

L'esistenza di tutte queste liste permette alcune scelte politiche di scelta; per esempio, supponiamo che una pagina debba essere portata dentro dal disco, ma la lista libera è vuota; il sistema è obbligato adesso a scegliere tra prendere una pagina pulita dalla lista standby (che invece potrebbe essere riportata indietro in seguito) o una pagina vuota dalla lista di pagine azzerate (buttando via il lavoro fatto per azzerarla). Quale è la soluzione migliore? Se la CPU è inattiva da molto ed il thread di azzeramento pagine va in esecuzione spesso, prendere una pagina azzerata risulta la scelta migliore perché non vi è scarsità di pagine azzerate; invece, se la CPU è spesso occupata ed il disco è inattivo, è meglio prendere una pagina dalla lista standby per evitare il costo di CPU di dover azzerare un pagina più tardi, se lo stack cresce.

Un altro rompicapo è il seguente: con quale grado di aggressività i demoni dovrebbero spostare le pagine dalla lista modificata alla lista standby? Avere intorno pagine pulite (che possono essere usate istantaneamente) è meglio che avere pagine sporche, ma una politica di pulizia aggressiva si traduce in più operazioni di I/O su disco e in probabilità che alcune delle pagine appena pulite possano essere riportate in un insieme di lavoro e sporcate nuovamente.

In generale, Windows 2000 risolve questi tipi di conflitti con euristiche complesse, stime, precedenti storici, regole di buonsenso, e parametri di controllo definiti dall'amministratore. In più, il codice è così complesso che i progettisti detestano toccare parti di esso per paura di rompere qualche legame in qualche posto del sistema che nessuno, di fatto, capisce più nella sua interezza.

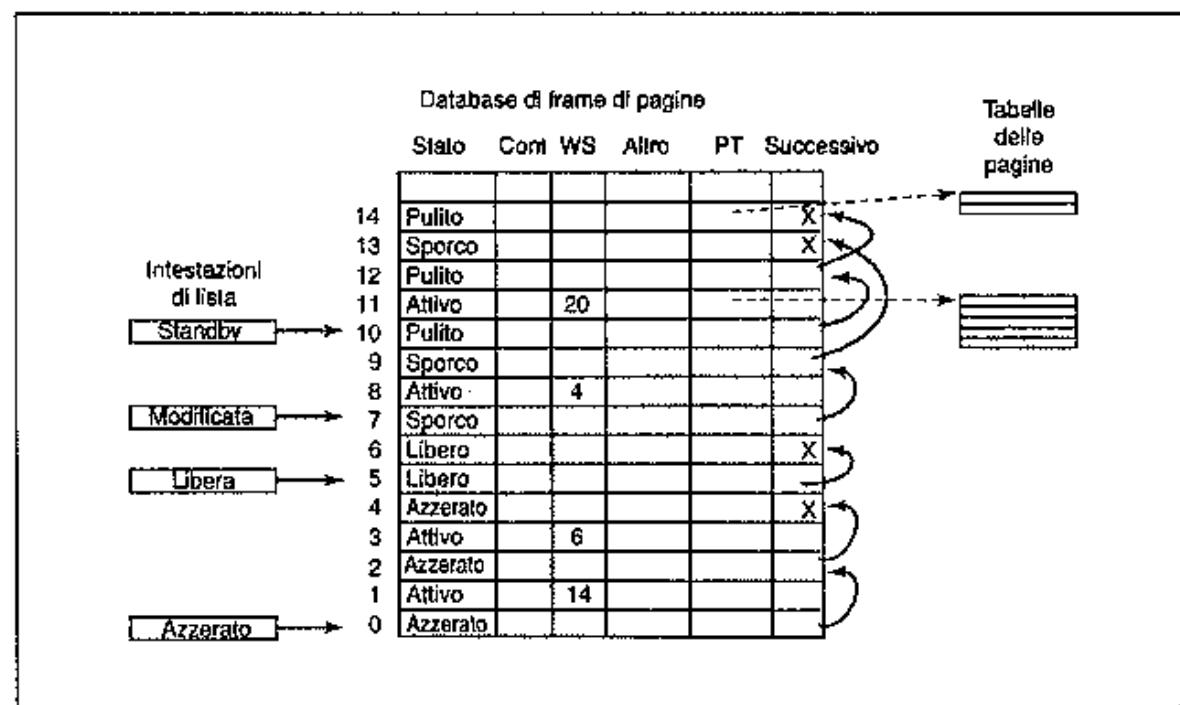


Figura 11.28 Alcuni dei campi più importanti nel database dei frame di pagina per una pagina valida.

Per mantenere traccia di tutte le pagine e di tutte le liste, Windows mantiene un database di frame di pagine, con tanti campi dato quante sono le pagine in RAM, come mostrato in Figura 11.28. Questa tabella è indicizzata da un numero fisico di frame di pagina; i dati sono di lunghezza fissa, ma si usano diversi formati per diversi tipi di campi dato (ad esempio, valido oppure invalido). I campi dato validi contengono lo stato della pagina, e un contatore di quante tabelle di pagina puntino alla pagina, in modo che il sistema possa capire quando la pagina non è più in uso. Le pagine che sono in un insieme di lavoro sanno quale sia questo insieme; vi è anche un puntatore alla tabella di pagina che punta alla pagina, se ve ne è una (le pagine condivise sono gestite in modo speciale), un collegamento alla pagina successiva sulla lista (se presente) e vari altri campi e flag, come lettura in corso, scrittura in corso, eccetera.

Tutto sommato, il gestore della memoria è un sottosistema fortemente complesso con molte strutture dati, algoritmi ed euristiche, che cerca di autoregolarsi, ma vi sono anche molte "manopole" che gli amministratori possono stringere per influenzare le prestazioni del sistema. Alcune di queste manopole ed i contatori associati possono essere visti usando strumenti nei vari pacchetti menzionati in precedenza. Probabilmente la cosa più importante da ricordare qui è che la gestione della memoria nei sistemi reali è molto di più di un semplice algoritmo di paginazione come l'algoritmo dell'orologio o quello dell'invecchiamento.

## 11.6 Input/Output in Windows 2000

L'obiettivo del sistema di I/O di Windows 2000 è di fornire una struttura per la gestione efficiente di una larghissima varietà di dispositivi di I/O; i dispositivi di input oggi comprendono vari tipi di tastiere, mouse, touch pad, joystick, scanner, macchine fotografiche digitali, fotocamere televisive, lettori di codici a barre, microfoni, e altra ferraglia da laboratorio. I dispositivi di output odierni comprendono monitor, stampanti, plotter, beamer, masterizzatori, e schede audio. I dispositivi di salvataggio comprendono floppy disk, hard disk IDE e SCSI, CD-ROM, DVD, Zip driver, registratori a nastro. Infine, altri dispositivi comprendono orologi, reti, telefoni, e videocamere. Non ci sono dubbi che molti nuovi dispositivi di I/O saranno inventati negli anni a venire, così Windows 2000 è stato progettato con una struttura generale alla quale potranno essere facilmente collegati i nuovi dispositivi; la sezione successiva esaminerà qualcuna delle peculiarità relative all'I/O.

### 11.6.1 Concetti fondamentali

Il gestore di I/O è intimamente connesso con il gestore del plug-and-play. L'idea alla base del plug-and-play è quella di elencare i bus; molti bus, tra cui quelli delle schede PC, PCI, USB, IEEE 1394, e SCSI, sono stati progettati in modo che il gestore del plug-and-play possa inviare richieste ad ogni slot, e chiedere al dispositivo di identificarsi. Avendo scoperto cosa c'è dall'altra parte, il gestore del plug-and-play alloca le risorse hardware, come i livelli di interruzioni, localizza i driver appropriati, e li carica in memoria; al momento del caricamento di ogni dispositivo, si crea per esso un oggetto driver. Per alcuni bus, come SCSI, la ricerca avviene solo al momento del boot di sistema, per altri bus come USB e IEEE 1394, può succedere in qualsiasi momento, richiedendo uno stretto contatto tra il gestore del plug-and-play, il driver del bus (che fa effettivamente l'elencazione), e il gestore di I/O.

Il gestore di I/O è anche strettamente connesso con il gestore dell'alimentazione; que-

st'ultimo può mettere il calcolatore in uno qualsiasi dei sei stati, approssimativamente descritti di seguito:

1. Pienamente operativo.
2. Sleep-1: Alimentazione della CPU ridotta, RAM e cache attive; risveglio istantaneo.
3. Sleep-2: CPU e RAM attive, cache della CPU spenta, può continuare dal contatore di programma corrente.
4. Sleep-3: CPU e cache spente; RAM accesa, riavvio da un indirizzo fissato.
5. Ibernato: CPU, cache e RAM spente; riavvio da un file salvato su disco.
6. Spento: Tutto spento; reboot totale necessario.

I dispositivi di I/O possono essere anche essi in vari stati di gestione dell'alimentazione; la loro accensione e spegnimento è gestita dal gestore dell'alimentazione insieme con il gestore dell'I/O. Da notare che gli stati dal 2 al 6 sono usati solo quando la CPU è stata inattiva per un intervallo di tempo più o meno lungo.

Abbastanza sorprendentemente, tutti i file system sono tecnicamente driver di I/O; le richieste di blocchi dati dei processi utente sono inviate inizialmente al gestore della cache, ma se il gestore della cache non può soddisfarle, fa in modo che il gestore di I/O chiami il driver del file system opportuno per recuperare dal disco il blocco dati richiesto.

Una funzionalità interessante di Windows 2000, è il supporto per **dischi dinamici**. Questi dischi possono occupare partizioni multiple e persino dischi multipli, e possono essere riconfigurati al volo, senza neanche bisogno di un riavvio del sistema. In questo modo, i volumi logici non sono vincolati ad una singola partizione, e neppure ad un singolo disco, cosicché un solo file system può gestire più dispositivi in modo trasparente.

Un altro interessante aspetto di Windows 2000 è il suo supporto per l'I/O asincrono: un thread può lanciare una operazione di I/O e continuare la sua esecuzione in parallelo con l'I/O. Questa funzionalità è particolarmente importante sui server. Ci sono vari modi con i quali un thread può essere avvisato del completamento dell'I/O: un modo è specificare un oggetto evento nel momento in cui viene fatta la chiamata, e poi attendere eventualmente su questo. Un altro è specificare una coda in cui il sistema inserirà un evento di completamento quando l'I/O è completato. Un terzo è fornire una procedura per la chiamata di ritorno (callback) che il sistema chiama quando l'I/O è completato.

## 11.6.2 Chiamate API di Input/Output

Windows 2000 ha oltre 100 API separate per una ampia varietà di dispositivi I/O, compresi i mouse, le schede sonore, i telefoni, i dispositivi a cassetta, eccetera; ma probabilmente il dispositivo più importante è il sistema grafico, per il quale vi sono migliaia di chiamate API Win32. Nella Sezione 5.7.3 avevano incominciato la nostra discussione sul sistema grafico di Windows; qui la continuiamo, citando un po' di nuove categorie API Win32, ognuna della quali ha molte chiamate. Un breve riassunto delle categorie è dato in Figura 11.29; come si è detto nel Capitolo 5, sono stati scritti parecchi libri di 1500 pagine sulla porzione grafica dell'API Win32.

Esistono chiamate di Win32 per creare, distruggere e gestire finestre; queste ultime hanno un vasto numero di stili ed opzioni che possono essere specificate, come titoli, bordi, colori, dimensioni, e barre di scorrimento. Le finestre possono essere fisse o mobili, di dimensione costante o ridimensionabili; si possono richiedere le loro proprietà, e si possono spedire loro dei messaggi.

Molte finestre contengono menu, e quindi vi sono chiamate Win32 per creare e can-

| Gruppo API              | Descrizione                                                            |
|-------------------------|------------------------------------------------------------------------|
| Gestione delle finestre | Crea distrugge e gestisce le finestre                                  |
| Menu                    | Crea distrugge ed aggiunge oggetti a menu e barre dei menu             |
| Finestre di dialogo     | Apri finestre di dialogo e raccolte informazioni                       |
| Disegno                 | Mostra punti, linee e figure geometriche                               |
| Testo                   | Mostra del testo con un certo carattere, una certa dimensione e colore |
| Icone e bitmap          | Posizionamento di bitmap ed icone sullo schermo                        |
| Colori e tavolozze      | Gestisce l'insieme di colori disponibile                               |
| La clipboard            | Passa una informazione da una applicazione ad un'altra                 |
| Input                   | Riceve informazioni da mouse e tastiera                                |

Figura 11.29 Alcune categorie di chiamate API Win32.

cellare menu e barre di menu; si possono rendere visibili e rimuovere i menu dinamici, e gli elementi dei menu possono essere evidenziati, fatti sbiadire e messi in cascata.

Si fanno comparire finestre di dialogo per comunicare qualche evento all'utente o per porgli una domanda; esse possono contenere bottoni, indicatori, o campi di testo da completare. Si possono associare suoni alle finestre di dialogo, per esempio nel caso di messaggi di avvertimento.

Sono disponibili centinaia di funzioni di disegno e coloritura, dalla possibilità di definire un singolo pixel fino a creare regioni complesse con operazioni di riempimento. Vengono fornite molte chiamate per disegnare linee e figure geometriche chiuse di vario tipo, con controllo dettagliato sulla tessitura, i colori, le dimensioni e molti altri attributi.

Un altro gruppo di chiamate sono quelle legate alla visualizzazione del testo; di fatto, la chiamata di visualizzazione del testo, `TextOut`, è banale; la complessità arriva dalla gestione del colore, del tipo di carattere, della dimensione del carattere, dei glifi, dello spazio orizzontale tra le lettere in una parola, e di altri dettagli tipografici. Fortunatamente, la rasterizzazione del testo (conversione in mappa di bit) si fa generalmente in modo automatico.

Le mappe di bit sono piccoli blocchi rettangolari di pixel che possono essere posti sullo schermo usando la chiamata `Win32 BitBlt`; esse sono utilizzate per le icone e occasionalmente per il testo. Per creare, distruggere e gestire gli oggetti icona sono disponibili varie chiamate.

Molti schermi utilizzano una modalità colore con solo 256 o 65536 dei  $2^{24}$  colori possibili, per rappresentare ogni pixel con 1 o 2 byte, rispettivamente; in questi casi, ci vuole una tavolozza dei colori per determinare quali sono i 256 o 65536 colori disponibili. Le chiamate in questo gruppo creano, distruggono e gestiscono le tavolozze, selezionano il colore disponibile più simile a quello richiesto, e cercano di rendere i colori a schermo il più possibile simili a quelli delle stampanti a colori.

Molti programmi di Windows 2000 permettono all'utente di selezionare dei dati (ad esempio una porzione di testo, una parte del disegno, un insieme di celle in un foglio elettronico), e metterli in un pannello di memoria (clipboard), che permette poi di incollare la parte copiata in un'altra applicazione; la clipboard generalmente si usa per questa trasmissione. Sono definiti molti formati di clipboard, tra cui il testo, le mappe di bit, gli oggetti e i metafile; questi ultimi sono insiemi di chiamate Win32 che, una volta eseguite, disegnano qualcosa, permettendo di tagliare ed incollare sezioni di un disegno. Que-

sto gruppo di chiamate pone le cose in una clipboard, tira fuori gli oggetti dalla clipboard e in generale li gestisce.

Infine, arriviamo all'input: non vi sono chiamate Win32 per applicazioni GUI per la lettura dell'input da tastiera, perché le applicazioni GUI sono guidate dagli eventi: il programma principale si compone di un grande loop che recupera messaggi di input. Quando un utente digita qualcosa di interessante, si spedisce un messaggio al programma per dirgli che sta arrivando qualcosa. Invece, vi sono chiamate relative al mouse, come leggere la sua posizione ( $x, y$ ) e lo stato dei suoi bottoni. Alcune chiamate di input sono in effetti chiamate di output, come ad esempio selezionare l'icona del cursore del mouse e muoverlo sullo schermo (fondamentalmente, questo è un output sullo schermo); ma le applicazioni non GUI possono leggere da tastiera.

### 11.6.3 Implementazione dell'I/O

Potremmo procedere più o meno all'infinito con le chiamate grafiche Win32, ma adesso è il momento di guardare come il gestore di I/O implementa la grafica e altre funzioni di I/O; la funzione principale del gestore di I/O è creare una struttura nella quale possano operare diversi dispositivi di I/O. La struttura di base è un insieme di procedure, indipendenti dal dispositivo, per certi aspetti di I/O, più un insieme di driver di dispositivi caricati per comunicare con i dispositivi.

### 11.6.4 I driver di dispositivi

Per essere certi che i driver di dispositivi lavorino bene con Windows 2000, Microsoft ha definito un **Modello di driver Microsoft (Microsoft Driver Model)**, al quale i driver di dispositivi dovrebbero essere conformi. In più, fornisce un pacchetto di strumenti che è stato progettato per aiutare i programmatori di driver a produrre driver conformi; in questa sezione cercheremo di esaminare brevemente questo modello. I driver conformi devono soddisfare i seguenti requisiti, ma anche altri:

1. Gestire le richieste di I/O che sopraggiungono, che arrivano in formato standard.
2. Basarsi su oggetti, come il resto di Windows 2000.
3. Fare in modo che i dispositivi plug-and-play vengano aggiunti e rimossi in modo dinamico.
4. Permettere la gestione dell'alimentazione, se applicabile.
5. Essere configurabili in termini di utilizzo di risorse.
6. Essere rientranti per l'utilizzo in caso di multiprocessori.
7. Essere portabili tra Windows 98 e Windows 2000.

Le richieste di I/O sono passate ai driver come pacchetti standardizzati chiamati IRP (I/O Request Packet – Pacchetti di richiesta I/O); i driver conformi ai requisiti devono essere in grado di gestire questi pacchetti. I driver devono essere basati su oggetti nel senso che devono supportare una lista specifica di metodi, che il resto del sistema può chiamare; essi devono inoltre essere in grado di gestire altri oggetti Windows 2000, quando un gestore di oggetto lo richiede.

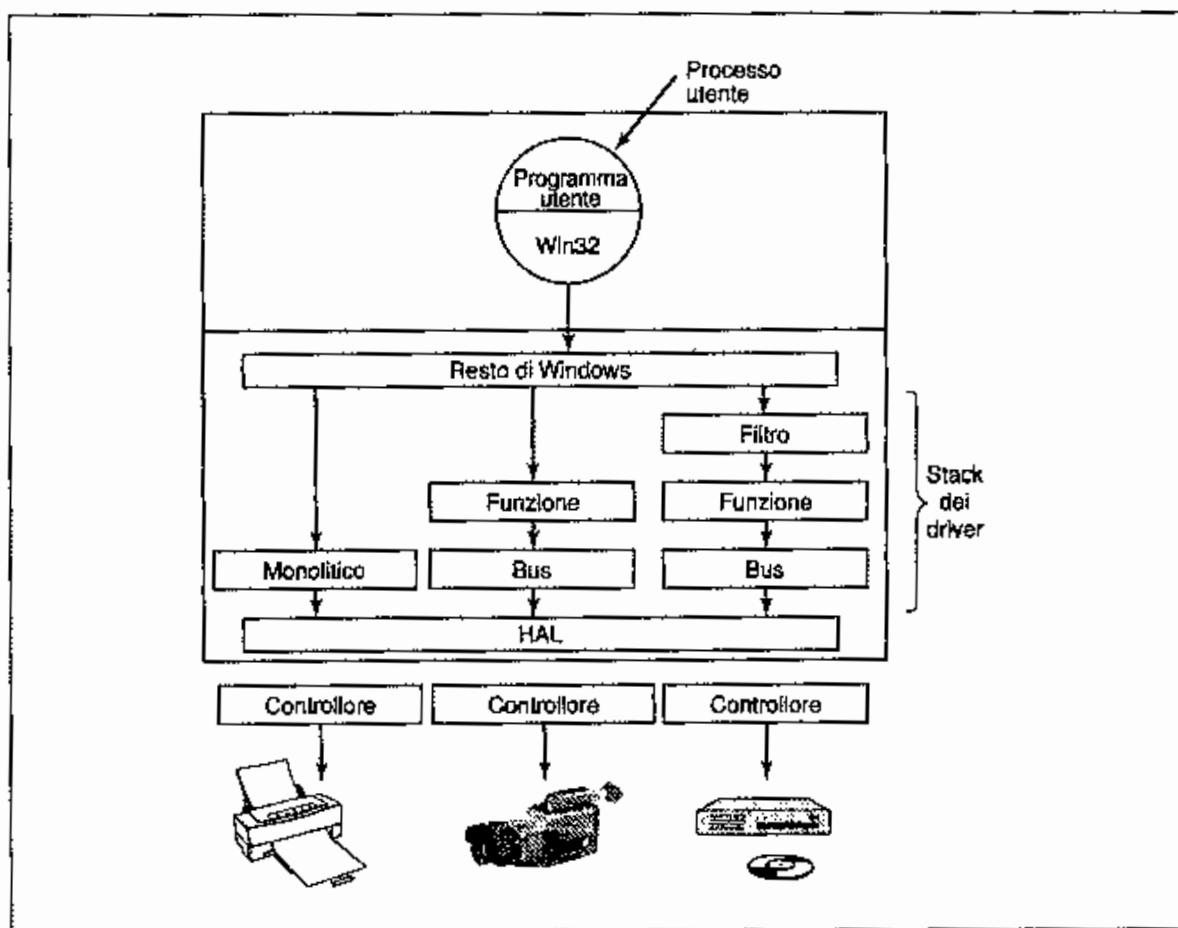
I driver conformi devono supportare in modo completo il plug-and-play, il che vuol dire che se un dispositivo gestito da un driver viene improvvisamente aggiunto o rimosso dal sistema, il driver deve essere pronto ad accettare questa informazione e ad agire su

di essa, anche nel caso in cui il dispositivo a cui si accede viene di colpo rimosso. Si deve anche supportare la gestione dell'alimentazione per i dispositivi per i quali è rilevante: per esempio, se il sistema decide che è il momento di passare in modalità di ibernazione a bassa alimentazione, tutti i dispositivi devono essere in grado di farlo per risparmiare energia; essi devono anche potersi svegliare quando gli viene richiesto.

I driver devono essere configurabili, il che vuol dire non avere alcuna ipotesi predefinita su quali linee di interruzione o porte di I/O devono usare i dispositivi; per esempio, la porta della stampante sul PC IBM e sui suoi successori sta all'indirizzo 0x378 da più di vent'anni, e non è probabile che cambi adesso, ma un driver di stampante che ha questo indirizzo scolpito all'interno non è conforme.

Essere adatti al multiprocessore è un requisito, visto che Windows 2000 è stato progettato per essere utilizzato su multiprocessori; concretamente questo requisito vuol dire che, mentre un driver sta girando ed elabora una richiesta a beneficio di una CPU, potrebbe venire una seconda richiesta da una diversa CPU, la quale potrebbe incominciare ad eseguire il codice del driver contemporaneamente alla prima. Il driver deve funzionare correttamente anche quando è eseguito in modo concorrente da due o più CPU, il che implica che tutte le strutture dati sensibili devono poter essere utilizzate solo all'interno di regioni critiche. Non è permesso supporre che non vi siano altre chiamate sino a quando quella corrente non è conclusa.

Infine i driver conformi devono funzionare sia su Windows 98 sia su Windows 2000; in ogni modo potrebbe essere necessario ricompilare il driver su ogni sistema, ed è per-



**Figura 11.30** Windows 2000 permette ai driver di essere impilati.

messo l'uso di direttive di precompilazione in C per isolare le dipendenze dalla piattaforma.

In UNIX, i driver sono localizzati usando i loro numeri di dispositivo principale; Windows 2000 usa uno schema differente. Al momento dell'avvio, o quando un nuovo dispositivo plug and play viene collegato al calcolatore, Windows 2000 lo rileva in modo automatico e chiama il gestore del plug-and-play. Il gestore richiede al dispositivo di scoprire l'identificativo del produttore ed il numero del modello; con queste informazioni esso guarda sull'hard disk in una certa directory per vedere se possiede il driver. Se non lo possiede, il gestore mostra una finestra di dialogo che chiede all'utente di inserire un dischetto o un CD-ROM con il driver; una volta localizzato il driver lo installa.

Ogni driver deve fornire un insieme di procedure che possono essere chiamate per richiedere i suoi servizi. La prima, chiamata *DriverEntry*, inizializza il driver; essa viene chiamata appena dopo che il driver è stato caricato, e può creare tabelle e strutture dati, ma non può ancora toccare il dispositivo. La procedura riempie anche qualche dato nei campi dell'oggetto driver creato dal gestore di I/O quando il driver è stato caricato; i campi nell'oggetto driver contengono puntatori a tutte le altre procedure che i driver devono fornire. In più, per ogni dispositivo controllato dal driver (ad esempio per ogni disco IDE controllato dal driver di disco IDE), viene creato un oggetto dispositivo (device object) e viene inizializzato perché punti all'oggetto driver. Questi oggetti driver sono inseriti in una directory speciale, \??; dato un oggetto dispositivo, l'oggetto driver può essere localizzato con facilità e quindi si possono chiamare i suoi metodi.

Una seconda procedura richiesta è *AddDevice* che è chiamata (dal gestore del plug and play) una sola volta per ogni dispositivo da aggiungere; fatto questo, il driver viene chiamato con il primo IRP che definisce il vettore di interruzioni e inizializza l'hardware. Altre procedure che i driver devono contenere sono: procedure di servizio di interruzioni, varie procedure per la gestione della temporizzazione, un cammino di I/O veloce, il controllo del DMA, un modo per cancellare le richieste correntemente in esecuzione e molte altre. Tutto sommato, i driver di Windows 2000 sono così complessi che su di loro sono stati scritti molti libri (Cant 1999; Oney 1999; Viscarola e Mason 1999).

Un driver di Windows 2000 può fare tutto il lavoro da solo, come fa il driver della stampante in Figura 11.30 (solo un esempio). D'altro canto, i driver possono anche essere impilati, che vuol dire che una richiesta può passare attraverso una sequenza di driver che svolgono ognuno una porzione di lavoro; due driver impilati sono illustrati in Figura 11.30.

Un uso comune per i driver impilati è di separare la gestione del bus dal lavoro funzionale di controllo del dispositivo. La gestione del bus PCI è abbastanza complicata a causa delle molte modalità di funzionamento e di transazione, e separando questo lavoro dalla parte specifica del dispositivo, i programmati di driver non hanno necessità di imparare il funzionamento del bus, ma possono utilizzare un driver di bus standard nella loro pila. Similmente, i driver USB e SCSI hanno una parte specifica del dispositivo ed una parte generica, con driver comuni utilizzati per la parte generica.

Un altro utilizzo che si fa dell'impilamento dei driver è la possibilità di inserire driver filtro nella pila; un driver filtro effettua alcune trasformazioni sui dati in una direzione o nell'altra; per esempio, un driver filtro potrebbe comprimere i dati nella direzione del disco o cifrare i dati nella direzione della rete. Porre il filtro a questo livello significa che né l'applicativo né il driver di dispositivo devono preoccuparsi di questo, e ciò funziona automaticamente con tutti i dati da (e per) i dispositivi.

## 11.7 Il file system di Windows 2000

Windows 2000 supporta diversi File System, tra i quali i più importanti sono: **FAT-16**, **FAT-32**, **NTFS (NT File System, file system di NT)**. FAT-16 è il vecchio file system di MS-DOS; utilizza indirizzi disco a 16 bit, che limitano la dimensione della partizione a 2GB. La FAT-32 utilizza indirizzi disco a 32 bit e supporta partizioni fino a 2 TB. NTFS è un nuovo file system sviluppato specificamente per Windows NT e trasportato su Windows 2000; utilizza indirizzi disco a 64 bit e può (teoricamente) supportare partizioni fino a  $2^{64}$  byte, sebbene altre considerazioni limitino le dimensioni a dei valori minori. Windows 2000 supporta anche file system di sola lettura per CD-ROM e DVD. È possibile (ed è comunemente diffuso) avere un sistema in esecuzione che accede a molti tipi di file system disponibili contemporaneamente.

In questo capitolo tratteremo il file system NTFS perché è un file system moderno, sgravato dalle necessità di essere completamente compatibile con il file system MS-DOS, che è basato sul file system CP/M progettato più di venti anni fa per dischetti a 8 pollici. I tempi sono cambiati e i dischetti a 8 pollici non sono più uno standard, e neanche il loro file system. Inoltre, NTFS differisce dal file system di UNIX sia in interfaccia utente sia in implementazione, il che lo rende un buon secondo caso di studio. NTFS è un sistema complesso ed ampio, e le limitazioni di spazio in questo libro ci inducono a non trattare in dettaglio tutte le sue funzionalità; tuttavia il materiale presentato di seguito dovrebbe darne una visuale ragionevolmente chiara.

### 11.7.1 Concetti fondamentali

I nomi dei file singoli in NTFS sono limitati a 255 caratteri; i cammini assoluti sono limitati a 32767 caratteri. I nomi di file sono in Unicode, permettendo alle persone che vivono in paesi che non utilizzano l'alfabeto latino (ad esempio, Grecia, Giappone, India, Russia e Israele) di scrivere nomi di file nel loro linguaggio nativo; per esempio `φιλε` è un nome perfettamente legale. NTFS distingue maiuscole e minuscole (quindi `foo` è diverso da `Foo` e da `FOO`), ma sfortunatamente l'API Win32 non le distingue perfettamente nei nomi di file e per nulla nei nomi di directory, quindi questo vantaggio viene perduto nei programmi che utilizzano Win32 (ad esempio per compatibilità con Windows 98).

Un file NTFS non è solo una sequenza lineare di byte come lo sono i file FAT-32 e UNIX; esso si compone di attributi multipli, ognuno dei quali è rappresentato da una sequenza di byte. La maggior parte dei file ha poche sequenze brevi, come il nome del file ed il suo ID di oggetto a 64 bit, più una sequenza lunga (senza nome) contenente i dati. Ad ogni modo, un file può anche avere due o più sequenze lunghe di dati; ogni sequenza ha un nome composto dal nome del file e dal nome della sequenza, separate da due punti, come ad esempio `foo:stream1`. Ogni sequenza ha la propria dimensione ed è bloccabile indipendentemente da tutte le altre; l'idea di sequenze multiple in un file è stata copiata dall'APPLE Macintosh nel quale i file hanno due sequenze, la biforcazione dati e la biforcazione risorse. Questo concetto è stato incorporato in NTFS per permettere ad un server NTFS di servire client Macintosh.

Le sequenze di file possono essere utilizzate per scopi che vanno al di là della compatibilità con i Macintosh; per esempio un programma per editare le foto potrebbe utilizzare una sequenza senza nome per l'immagine principale e una sequenza con nome per la relativa miniatura. Questo schema è più semplice del modo tradizionale di metterle (fotografia e miniatura) nello stesso file una dopo l'altra. Un altro uso delle sequenze è nella gestione della videoscrittura; questi programmi spesso fanno due versioni di un docu-

mento, una temporanea da usare durante la scrittura ed una finale per il salvataggio; se il file temporaneo fosse una sequenza con nome, e il file finale una sequenza senza nome, entrambe le versioni avrebbero contemporaneamente lo stesso nome di file, le stesse informazioni sulla sicurezza, timestamp eccetera, senza nessun lavoro extra.

La lunghezza massima di una sequenza è  $2^{64}$  byte. Per avere una minima idea di quanto è lunga una sequenza di  $2^{64}$  byte si immagini che la sequenza sia stata scritta in binario con gli 0 o 1 di ogni byte che occupano un millimetro di spazio; i  $2^{67}$  millimetri di lista-to sarebbero lunghi 15 anni luce, andando al di là del sistema solare fino ad Alfa Centauri e ritorno. I puntatori ai file sono usati per tenere traccia di dove si trova il processo in ogni sequenza, ed occupano 64 bit per gestire la massima lunghezza di una sequenza che è circa 18,4 exabyte.

Le chiamate di funzioni API Win32 per la manipolazione di file e directory sono molto simili alle controparti in UNIX, tranne che molte hanno più parametri, e che il modello di sicurezza è differente. Aprire un file restituisce un gestore, che è in seguito utilizzato per leggere e scrivere il file; per le applicazioni grafiche non esistono gestori di file predefiniti. Lo standard input, lo standard output e lo standard error devono essere acquisiti esplicitamente se necessario; ad ogni modo, in modalità console, essi vengono aperti automaticamente. Inoltre, Win32 possiede un certo numero di chiamate addizionali che non sono presenti in UNIX.

## 11.7.2 Le chiamate API del file system in Windows 2000

Le principali funzioni API Win32 per la gestione dei file sono elencate in Figura 11.31; ce ne sono molte di più, ma queste danno una prima impressione di quelle di base. Andiamo ad esaminare queste chiamate brevemente: `CreateFile` può essere utilizzata per creare un nuovo file e restituirci il gestore; può anche essere utilizzata per aprire file esistenti, poiché non c'è la funzione API `FileOpen`. Non abbiamo mostrato i parametri per le funzioni API perché sono molto voluminosi. Per esempio, `CreateFile` ha sette parametri, che sono sommariamente mostrati qui di seguito:

1. Un puntatore al nome del file da aprire o creare.
2. Un flag che dice se il file può essere letto, scritto o letto/scritto.
3. Un flag che dice se più processi possono aprire il file contemporaneamente.
4. Un puntatore al descrittore di sicurezza, che dice chi può accedere al file.
5. Un flag che dice cosa fare se il file esiste o non esiste.
6. Un flag avente a che fare con gli attributi come archiviazione, compressione, eccetera.
7. Il gestore di un file, gli attributi del quale vanno clonati per il nuovo file.

Le sei funzioni API che seguono in Figura 11.31 sono abbastanza simili alle corrispondenti chiamate di sistema UNIX; le ultime due permettono ad una regione di un file di essere bloccata o sbloccata per permettere ad un processo di avere garanzie di mutua esclusione su di esso.

Utilizzando queste funzioni API, è possibile scrivere procedure per copiare file, analoghe alla versione UNIX di Figura 6.5. Il frammento di codice (senza nessun controllo di errore) in Figura 11.32 è stato progettato copiando la nostra versione UNIX; in pratica, si potrebbe non dover programmare una funzionalità di copia file dal momento che `CopyFile` è una funzione API (che esegue qualcosa di molto simile a questo programma con una procedura di libreria).

Windows 2000 NTFS è un file system gerarchico, simile al file system di UNIX, ma

| Funzione API Win32 | UNIX   | Descrizione                                                 |
|--------------------|--------|-------------------------------------------------------------|
| CreateFile         | open   | Crea o apre un file esistente; restituisce il gestore       |
| DeleteFile         | unlink | Distrugge un file esistente                                 |
| CloseHandle        | close  | Chiude un file                                              |
| ReadFile           | read   | Legge i dati dal file                                       |
| WriteFile          | write  | Scrive i dati sul file                                      |
| SetFilePointer     | lseek  | Definisce il nuovo valore nel file del puntatore di file    |
| GetFileAttributes  | stat   | Restituisce le proprietà del file                           |
| LockFile           | fcntl  | Per fornire la mutua esclusione blocca una regione del file |
| UnlockFile         | fcntl  | Sblocca una regione del file precedentemente bloccata       |

**Figura 11.31** Le principali funzioni API Win32 per l'I/O dei file. La seconda colonna fornisce il più simile equivalente UNIX.

```
/* apre file per input e output */
inhandle = CreateFile("dato", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("nuovo", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

/* copia del file */
do {
 s = ReadFile(inhandle, buffer, BUF_SIZE, &cont, NULL);
 if (s && cont > 0) WriteFile(outhandle, buffer, cont, &ocnt, NULL);
} while (s > 0 && cont > 0);

/* chiude il file */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

**Figura 11.32** Un frammento di programma per copiare un file utilizzando le funzioni API di Windows 2000.

il separatore per la composizione dei nomi è \, invece di /: un fossile ereditato da MS-DOS. In NTFS è presente il concetto di directory di lavoro corrente, ed i nomi di percorso possono essere relativi ed assoluti. Sono supportati i collegamenti hardware e simbolici, i primi implementati con accessi multipli alle directory, come in UNIX, e gli ultimi implementati utilizzando analisi sintattica ripetuta (reparse, discusso più avanti in questo capitolo). In più sono supportate compressione, cifratura, e tolleranza ai guasti; queste funzionalità e la loro implementazione saranno discusse più avanti in questo capitolo.

Le più importanti funzioni API per la gestione delle directory sono date in Figura 11.33; di nuovo con le loro corrispondenze in UNIX. Le funzioni dovrebbero essere auto-esplicative.

### 11.7.3 Implementazione del file system di Windows 2000

NTFS è un file system altamente complesso e sofisticato; fu progettato ex-novo, piuttosto che cercando di migliorare il vecchio file system di MS-DOS. Di seguito esaminere-

mo un insieme delle sue funzionalità, partendo dalla sua struttura per poi spostarci al recupero dei nomi di file, compressione di file e cifratura di file.

## Struttura del file system

Ogni volume NTFS (ad esempio una partizione del disco) contiene file, directory, mappe di bit, ed altre strutture dati; esso è organizzato in una sequenza lineare di blocchi (cluster nella terminologia Microsoft) con dimensione del blocco fissata per ogni volume e variabile tra i 512 byte ed i 64KB, in base alla dimensione del volume. La maggior parte dei dischi NTFS usa blocchi di 4KB come compromesso tra blocchi grandi (a favore di un trasferimento efficiente) e blocchi piccoli (a favore di una minore frammentazione interna). I blocchi sono indirizzati tramite l'offset relativo dall'inizio del volume, usando numeri a 64-bit.

La struttura principale in ogni volume è MFT (Master File Table), che è una sequenza lineare di record di 1KB; ogni record MFT descrive un file o una directory, e contiene gli attributi del file, ad esempio il nome e il timestamp, e la lista di indirizzi disco dove i blocchi sono localizzati. Se un file è estremamente grande, talvolta è necessario usare due o più record MFT per essere in grado di contenere la lista di tutti i blocchi; in questo caso il primo MFT viene chiamato record di base (base record), e punta a tutti gli altri record. Questo schema ci riporta in un certo senso al CP/M dove ogni campo dato (entry) in una directory era chiamata estensione (extent). Una mappa di bit tiene traccia degli MFT liberi.

Un MFT è a sua volta un file e per questo può essere posto in un qualsiasi punto nel volume, eliminando quindi il problema di settori imperfetti nella prima traccia. In più, il file può crescere, se richiesto, fino ad un massimo di  $2^{48}$  record.

Un MFT è mostrato in Figura 11.34; ogni record MFT è composto da una sequenza di coppie (intestazione di attributo, valore). Ogni attributo inizia con una intestazione, che dice quale attributo sia e quanto è lungo il valore, poiché alcuni attributi di valore sono di lunghezza variabile, come il nome di file ed i dati. Se un valore di attributo è corto abbastanza da entrare nel record MFT, viene posto in esso; se è troppo lungo, viene posto altrove sul disco, e nel record MFT viene posto un puntatore ad esso.

I primi 16 record MFT sono riservati ai file di metadata NTFS, come mostrato in Figura 11.34; ogni record descrive un file normale che ha attributi e blocchi di dati, come un qualsiasi altro file. Ognuno di questi file ha un nome che comincia con un segno di dollaro (\$) per indicare che si tratta di un metadata; il primo record descrive il file MFT stesso, ed in particolare, esso rivela dove sono posti i blocchi del file MFT in

| Funzione API Win32  | UNIX    | Descrizione                                           |
|---------------------|---------|-------------------------------------------------------|
| CreateDirectory     | mkdir   | Crea una nuova directory                              |
| RemoveDirectory     | rmdir   | Rimuove una directory vuota                           |
| FindFirstFile       | opendir | Inizializza per leggere gli elementi di una directory |
| FindNextFile        | readdir | Legge il successivo elemento della directory          |
| MoveFile            | rename  | Sposta un file da una directory all'altra             |
| SetCurrentDirectory | chdir   | Cambia l'attuale directory di lavoro                  |

Figura 11.33 Le funzioni API Win32 più importanti per la gestione delle directory. La seconda colonna dà i corrispondenti in UNIX, quando questi esistono.

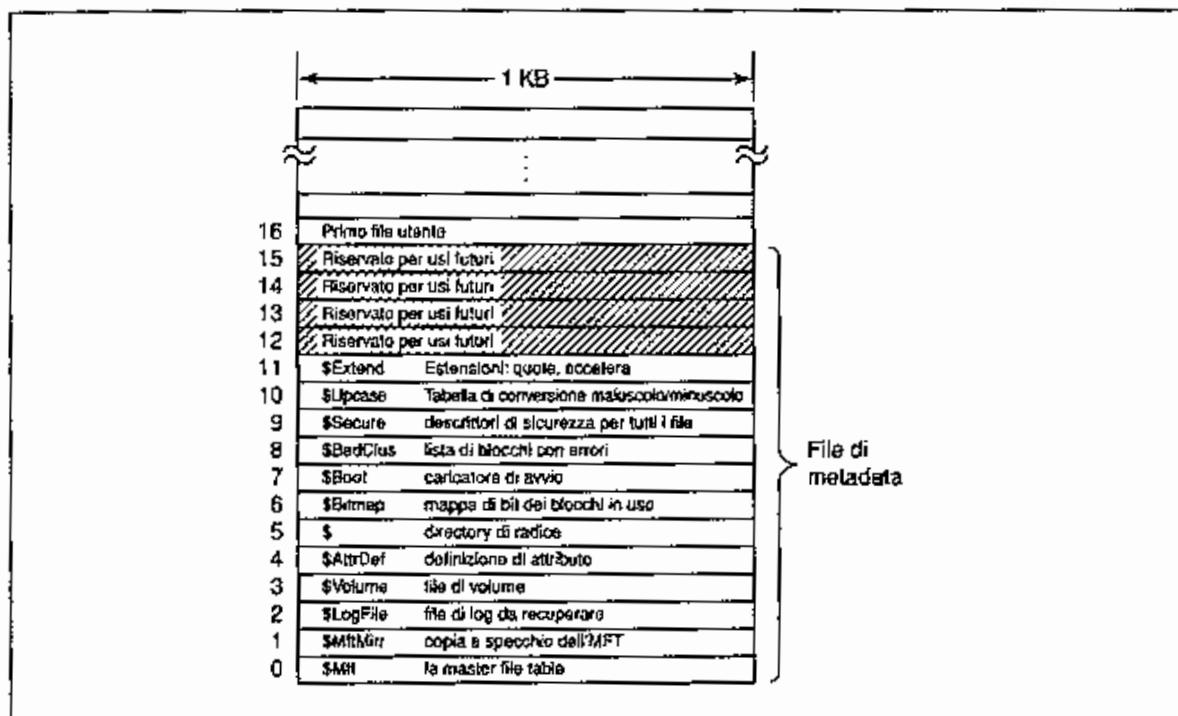


Figura 11.34 La Master File Table dell'NTFS.

modo che il sistema lo possa recuperare. Chiaramente, Windows 2000 necessita di un modo per trovare il primo blocco del file MFT, per trovare il resto delle informazioni di file system; e lo trova guardando nel blocco di avvio, dove il relativo indirizzo è stato posto al momento dell'installazione.

Il record 1 è un duplicato della prima parte di un file MFT; questa informazione è così preziosa che averne una seconda copia può essere importante, nel caso che il primo blocco di un MFT sia in qualche modo danneggiato. Il record 2 è un file di registrazione (log); quando vengono fatti cambiamenti strutturali al file system, come nel caso dell'aggiunta di una nuova directory e della rimozione di una directory esistente, l'azione viene registrata qui, prima di farla, in modo da migliorare le possibilità di un recupero corretto nel caso di un fallimento nel corso dell'operazione. Qui vengono anche registrati i cambiamenti ad attributi di file; in effetti, gli unici cambiamenti non registrati qui sono i cambiamenti ai dati utente. Il record 3 contiene informazioni sul volume, come la dimensione, l'etichetta, e la versione.

Come già detto prima, ogni record MFT contiene una sequenza di coppie (intestazione di attributo, valore); il file \$AttrDef è il posto in cui vengono definiti gli attributi, e le informazioni sul file si trovano nel record 4. Quindi si trova la directory di radice, che è un file, e può essere di lunghezza arbitraria; essa è descritta nel record 5.

Lo spazio libero su disco viene tracciato in una mappa di bit. La mappa di bit è essa stessa un file, ed i suoi attributi e indirizzi disco sono dati dal record MFT numero 6. Il record seguente punta al file di caricamento di bootstrap, mentre il record 8 viene usato per collegare tutti i blocchi mal funzionanti in modo da essere sicuri di non utilizzarli per alcun file. Il record 9 contiene le informazioni di sicurezza, ed il record 10 viene usato per la mappa dei caratteri maiuscoli. Per le lettere latine A-Z la mappa è ovvia (almeno per le persone che parlano lingue di base latina), ma la mappa di altri linguaggi, come il Greco, l'Armeno, o il Georgiano (la lingua parlata nello stato ex-URSS, da non confondere con la Georgia, USA), è meno ovvia per chi parla lingue a base lati-

| Attributo                   | Descrizione                                                                  |
|-----------------------------|------------------------------------------------------------------------------|
| Informazioni standard       | Bit di flag, timestamp, eccetera                                             |
| Nome del file               | Nome di file in Unicode; può essere ripetuto per i nomi in MS-DOS            |
| Descrittore di sicurezza    | Obsoleto. Adesso le informazioni di sicurezza si trovano in \$Extend\$Secure |
| Lista di attributi          | Locazione di MFT addizionali, se necessario                                  |
| ID di oggetto               | Identificatore 64-bit unico per questo volume                                |
| Punto di analisi sintattica | Usato per il montaggio e per i link simbolici                                |
| Nome di volume              | Nome di questo volume (usato solo in \$Volume)                               |
| Informazioni sul volume     | Versione del volume (usata solo in \$Volume)                                 |
| Radice dell'indice          | Usata per le directory                                                       |
| Allocazione di indice       | Usata per directory molto grandi                                             |
| Mappa di bit                | Usata per directory molto grandi                                             |
| Flusso di utilità di log    | Controlla gli accessi a \$LogFile                                            |
| Dati                        | Flusso di dati; può essere ripetuto                                          |

Figura 11.35 Gli attributi usati nei record MFT.

na, e quindi, questo file dice come avviene. Infine, il record 11 è una directory contenente file di vario tipo, per cose come le quote disco, identificativi di oggetti, punti di analisi sintattica e così via; gli ultimi 4 record MFT sono riservati ad usi futuri.

Ogni record MFT si compone di una intestazione del record seguita da una sequenza di coppie (intestazione di attributo, valore). L'intestazione del record contiene un numero magico per il controllo di validità, un numero di sequenza aggiornato ogni volta che il record viene riusato per un nuovo file, un contatore dei riferimenti al file, il numero effettivo dei byte nel record usato, un identificativo (indice, numero in sequenza) del record di base (usato solamente per i record di estensione), e qualche altro campo. Subito dopo l'intestazione del record vi è l'intestazione del primo attributo, quindi il valore del primo attributo, l'intestazione del secondo attributo, il valore del secondo attributo, e così via.

NTFS definisce 13 attributi che possono apparire in un record MFT, elencati in Figura 11.35. Ogni record MFT si compone di una sequenza di intestazioni ognuna delle quali identifica l'attributo relativo e, insieme ad una varietà di flag ed altre informazioni, fornisce la lunghezza e la locazione del campo valore. Di solito, i valori di attributo seguono direttamente le intestazioni ma, se il valore è troppo lungo per entrare nel record MFT, esso può essere collocato in un blocco disco separato. In tal caso, esso viene definito attributo non residente; l'attributo dato è un esempio ovvio. Alcuni attributi, ad esempio il nome, possono essere ripetuti ma, nel record MFT, tutti gli attributi devono apparire in un ordine fisso. Le intestazioni per gli attributi residenti sono lunghe 24 byte; invece, per gli attributi non residenti, sono più lunghe dato che contengono informazioni riguardo a dove poter ritrovare nel disco gli attributi relativi.

Il campo di informazione standard contiene il proprietario del file, le informazioni di sicurezza, i timestamp necessari per POSIX, il contatore dei link fisici, i bit sola-lettura e archivio, eccetera; è un campo di lunghezza fissa che è sempre presente. Il nome del file è di lunghezza variabile in Unicode; per rendere i file con i nomi non compatibili con MS-DOS disponibili per i vecchi programmi a 16-bit, i file possono avere anche un nome a 8+3 caratteri stile MS-DOS. Se il nome primario è compatibile con le regole 8+3 di MS-DOS, il secondo nome non viene utilizzato.

In NT 4.0, le informazioni di sicurezza potevano andare in un attributo, ma in Windows 2000 vanno tutte in un singolo file, in modo che molti file possano condividerle. La lista di attributi è necessaria nel caso gli attributi non entrino perfettamente nel record MFT; questo attributo (la lista) dice quindi dove trovare i record di estensione. Ogni campo dato nella lista contiene un indice a 48-bit nel MFT che definisce dove si trova il record di estensione, ed un numero in sequenza di 16-bit per permettere la verifica che il record di estensione ed i record di base coincidono.

L'attributo di identificazione di oggetto fornisce al file un nome unico, che talvolta è necessario internamente. Il punto di analisi sintattica ripetuta (reparse) informa la procedura di analisi sintattica del nome del file di compiere qualche operazione speciale; tale meccanismo viene utilizzato per il montaggio e per i collegamenti simbolici. I due attributi di volume sono utilizzati solo per l'identificazione del volume, ed i tre attributi successivi hanno a che fare con il modo di implementare le directory: quelle piccole sono solo una lista di file, ma quelle grandi sono implementate usando i B+ tree. L'attributo per l'utilità di registrazione di sequenza viene usato dal file system di cifratura.

Infine, arriviamo all'attributo che tutti stavamo aspettando: i dati. Il nome del flusso dati, se presente, va nella sua intestazione di attributo; quindi l'intestazione è o una lista di indirizzi disco che servono per recuperare i blocchi che rappresentano il file, o per file di poche centinaia di byte (e ce ne sono molti di questo tipo) il file stesso. Un file che viene posto all'interno del record MFT relativo viene chiamato file immediato (*immediate file*) (Mullender e Tanenbaum, 1987).

Certamente, nella maggior parte dei casi, il file non entra perfettamente nel record del MFT e quindi, in genere, questo attributo è non residente; vediamo adesso come NTFS tiene traccia della locazione degli attributi non residenti, in particolare dei dati.

Il modello per mantenere traccia dei blocchi disco è assegnarli a sequenze di blocchi consecutivi, se possibile, per ragioni principalmente di efficienza; per esempio, se il primo blocco logico di un file è posto nel blocco 20 del disco, in seguito il sistema cercherà di piazzare il secondo blocco logico nel blocco 21, il terzo nel blocco 22, e così via. Un

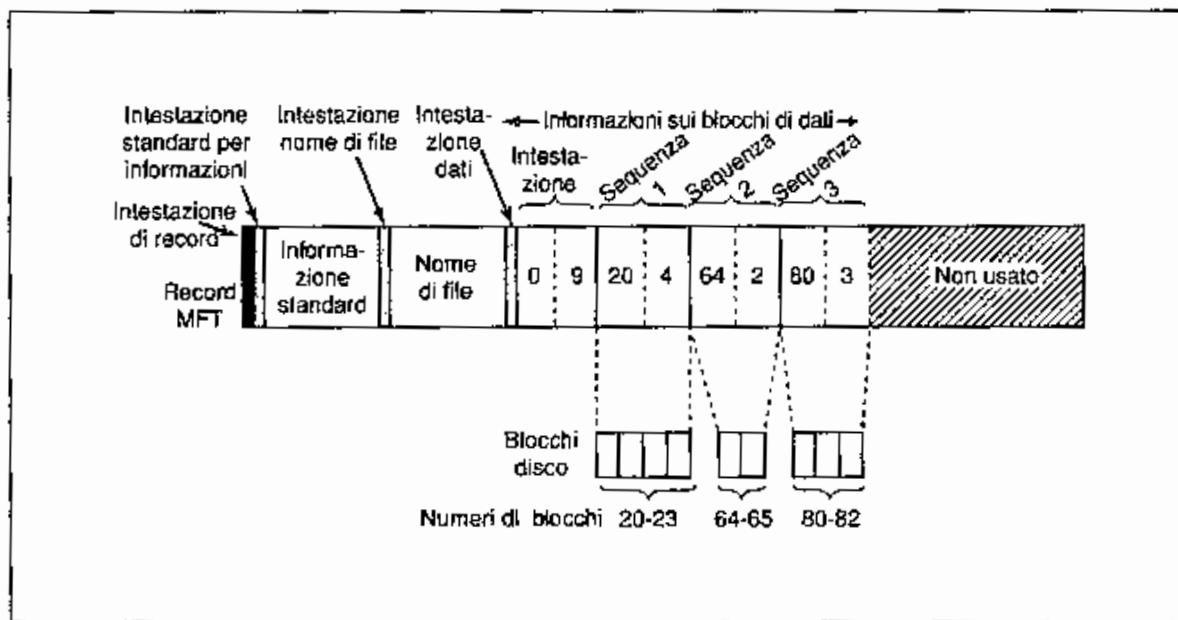


Figura 11.36 Un record MFT per un file a nove blocchi con tre sequenze.

modo per raggiungere queste sequenze è quello di allocare parecchi blocchi all'istante, se possibile.

I blocchi in un file sono descritti da una sequenza di record, ognuno rappresentante una sequenza di blocchi logicamente contigui; per un file senza buchi, vi sarà uno solo di questi record, ed i file che sono scritti in ordine dall'inizio alla fine appartengono a questa categoria. Per un file con un buco (ad esempio, se ha definiti solo i blocchi 0-49 e 60-79), vi saranno due record: tale file potrebbe essere stato prodotto scrivendo i primi 50 blocchi, quindi cercando il blocco 60 e scrivendo i successivi 20 blocchi. Quando si legge un buco, tutti i byte mancanti sono zero.

Ogni record inizia con una intestazione che da l'offset del primo blocco nel file, seguono gli offset del primo blocco non coperti dal record. Nell'esempio di sopra, il primo record dovrebbe avere una intestazione (0, 50) e fornire gli indirizzi di questi 50 blocchi; il secondo dovrebbe avere una intestazione (60, 80) e fornire gli indirizzi dei 20 blocchi.

Ogni intestazione è seguita da una o più coppie, che forniscono un indirizzo disco e una lunghezza di sequenza; l'indirizzo disco è l'offset del blocco disco dall'inizio della partizione, la lunghezza di sequenza è il numero di blocchi nella stessa. Nel record vi saranno tutte le coppie necessarie; l'utilizzo di questo schema per un file di nove blocchi con tre sequenze è illustrato in Figura 11.36.

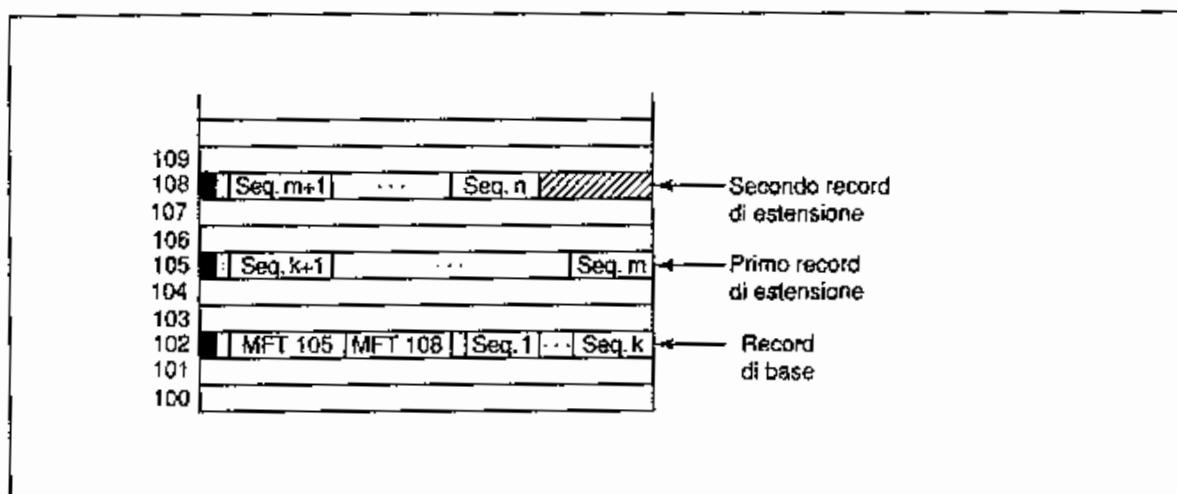
Nella figura abbiamo un record MFT per un file corto (corto in questo caso vuole dire che tutte le informazioni circa i blocchi del file si trovano all'interno del record MFT); esso si compone delle tre sequenze di blocchi consecutivi sul disco, di cui la prima è rappresentata dai blocchi 20-23, la seconda dai blocchi 64-65 e la terza dai blocchi 80-82. Ognuna di queste sequenze viene registrata nel record MFT sotto forma di una coppia (indirizzo disco, contatore di blocco). Il numero di sequenze presenti dipende da quanto è stato bravo l'allocatore di blocchi disco, al momento in cui il file è stato creato, a trovare blocchi consecutivi. Per un file con  $n$  blocchi il numero di sequenze potrebbe essere un qualsiasi valore tra 1 ed  $n$ .

Parecchi commenti sono necessari a questo punto: in primo luogo, non esiste alcun limite superiore alla dimensione di file che possono essere rappresentati in questo modo; in assenza di compressione di indirizzi, ogni coppia richiede due numeri a 64-bit (per coppia) per un totale di 16 byte. Ad ogni modo, una coppia potrebbe rappresentare un milione o più blocchi disco consecutivi; in effetti, un file di 20MB composto da 20 sequenze separate di 1 milione di blocchi da 1KB ciascuna sta perfettamente in un record MFT, cosa che non succede per un file di 60KB sparso in 60 blocchi isolati.

In secondo luogo, mentre esiste un modo immediato per rappresentare ogni coppia con  $2 \times 8$  byte, è a disposizione un metodo di compressione per ridurre la dimensione delle coppie al di sotto dei 16 byte: molti indirizzi disco hanno molti zeri non significativi in testa, che possono essere omessi, purché l'intestazione dei dati dica quanti sono stati omessi, cioè, quanti byte sono veramente utilizzati per l'indirizzo. Esistono anche altri metodi di compressione, per cui in pratica, le coppie occupano spesso solo 4 byte.

Il nostro primo esempio è stato semplice: tutte le informazioni entravano nel record MFT, ma cosa succede se il file è così grande, o talmente frammentato, che le informazioni sui blocchi non entrano perfettamente nel record MFT? La risposta è semplice: si utilizzano due o più record MFT per il file. In Figura 11.37 vediamo un file per il quale il record di base MFT è il record 102; esso possiede troppe sequenze per un unico record, quindi si calcola quanti record estensione occorrono, diciamo due, e si inseriscono i relativi indici nel record di base; il resto del record viene utilizzato per inserire le prime  $k$  sequenze di dati.

Si osservi che la Figura 11.37 contiene alcune ridondanze; in teoria, potrebbe non essere necessario specificare la fine di una sequenza di sequenze in quanto questa informazione può essere calcolata dalla coppia di sequenze. La ragione di questa "sovra-specifi-



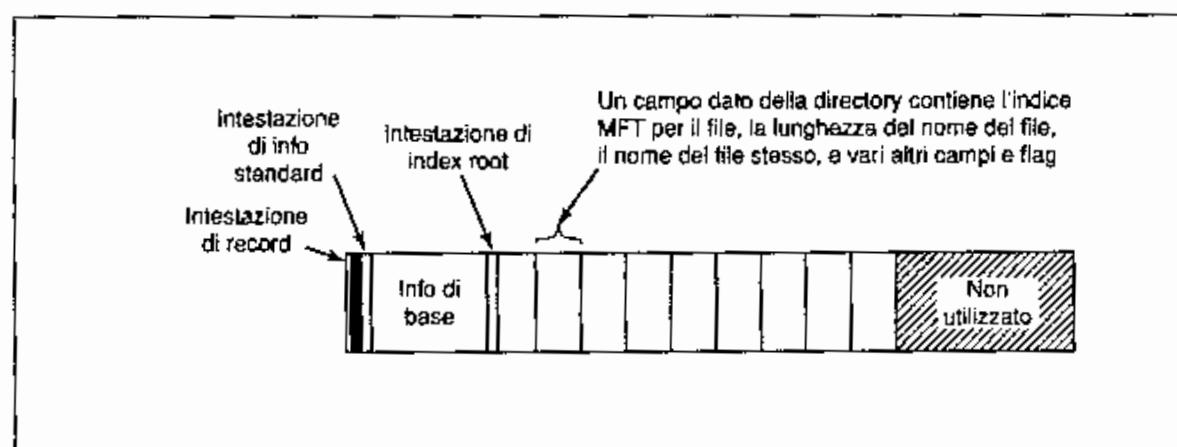
**Figura 11.37** Un file che richiede tre record MFT per immagazzinare tutte le sequenze.

cazione" è di rendere il ritrovamento di blocchi più efficiente: per trovare il blocco ad un determinato offset, è necessario solamente esaminare le intestazioni dei record, non le coppie di sequenze.

Quando è stato utilizzato tutto lo spazio nel record 102, l'immagazzinamento delle sequenze continua nel record MFT 105, in cui vengono immagazzinate le sequenze fino a quando è possibile. Quando anche questo record è pieno, il resto delle sequenze è inserito nel record MFT 108. In questo modo possono essere utilizzati molti record MFT per gestire grandi file frammentati.

Sorge un problema nel caso in cui siano necessari così tanti record MFT che la relativa lista non entra tutta in un solo record MFT di base. Esiste una soluzione anche a questo problema: la lista dei record MFT estensioni viene fatta in modo non residente (cioè, immagazzinata su disco invece che nel record MFT di base); in questo modo può aumentare a piacere.

Un campo dato MFT per una piccola directory è mostrato in Figura 11.38. Il record contiene un numero di campi dato directory, ognuno descrivente un file o una directory; ogni campo dato ha una struttura a lunghezza fissata seguita da un nome di file a lunghezza variabile. La parte fissata contiene l'indice del campo dato MFT per il file, la lun-



**Figura 11.38** Il record MFT per una piccola directory.

ghezza del nome del file, ed una varietà di altri campi e flag. La ricerca di un campo dato in una directory consiste nell'esaminare a turno tutti i nome di file.

Le directory di una certa dimensione usano un formato differente; invece di elencare i file linearmente, viene utilizzato un B+ tree per fare una ricerca alfabetica e per rendere semplice inserire nuovi nomi nella directory nel posto corretto.

## Recupero di nomi di file

Adesso possediamo abbastanza conoscenze per vedere con avviene un recupero di nomi di file. Quando un programma utente vuole aprire un file, esso fa tipicamente una chiamata come

```
CreateFile("C:\maria\web.htm", ...)
```

Questa chiamata utilizza la libreria condivisa a livello utente, *kernel32.dll*, dove \?? viene posto in testa al nome dato, ottenendo

```
\??\C:\maria\web.htm
```

questo è il nome di file passato alla chiamata di sistema *NtFileCreate*.

Successivamente il sistema operativo incomincia la ricerca alla radice dello spazio dei nomi del gestore di oggetti (vedi Figura 11.12), quindi guarda la directory \?? per trovare C:, che troverà. Questo file è un collegamento simbolico ad un'altra parte dello spazio dei nomi del gestore di oggetti, la directory \Device; il collegamento, tipicamente, finisce ad un oggetto il cui nome è qualcosa come \Device\Harddisk Volume1, che corrisponde alla prima partizione del primo hard disk. Da questo oggetto è possibile determinare quale MFT utilizzare, cioè quella della partizione stabilita; questi passi sono mostrati in Figura 11.39.

L'analisi sintattica del nome del file continua fino alla directory radice, i cui blocchi possono essere trovati dal campo dato 5 nell'MFT (vedi Figura 11.34). La stringa "maria" viene ricercata nella directory radice, che restituisce un indice nel record MFT per la directory *maria*; questa directory viene quindi esaminata alla ricerca della stringa "web.htm" e, se la ricerca ha successo, il risultato è la creazione di un nuovo ogget-

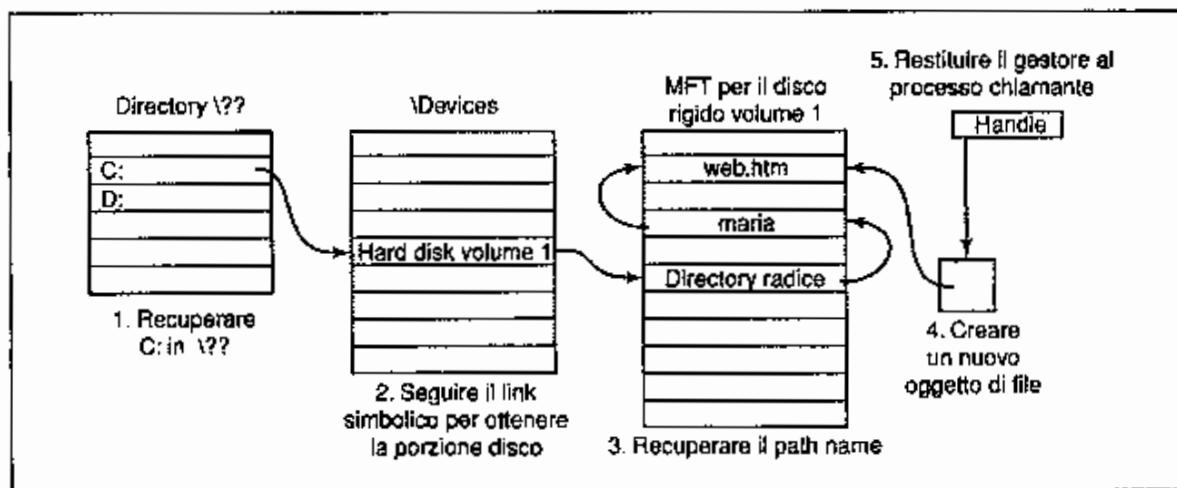


Figura 11.39 I passi necessari per recuperare il file C:\maria\web.htm.

to da parte del gestore di oggetti. L'oggetto, che è senza nome, contiene l'indice del record MFT per il file, ed il gestore di questo oggetto viene restituito al processo chiamante. Nelle successive chiamate a `ReadFile`, viene fornito il gestore, che permette al gestore di oggetti di trovare l'indice e quindi il contenuto del record MFT per il file. Se un thread in un secondo processo apre nuovamente il file, esso riceve un gestore per un nuovo oggetto file.

In aggiunta ai file e directory regolari, NTFS supporta collegamenti fisici (hard link) in senso UNIX, ed anche collegamenti simbolici utilizzanti il meccanismo di `reparse point` (punti di analisi sintattica). È possibile marcare un file o una directory come punto di analisi sintattica, ed associarvi un blocco di dati; quando il file o la directory viene incontrata durante l'analisi sintattica del nome di un file, si attiva una trap ed il blocco di dati viene interpretato. Si possono fare varie cose, compreso il fatto di ridirigere la ricerca ad una parte diversa della gerarchia di directory o anche ad una differente partizione. Questo meccanismo viene utilizzato per supportare sia i collegamenti simbolici sia i file system montati.

## Compressione di file

NTFS supporta una compressione trasparente di file. Un file può essere creato in modo compresso, che vuole dire che NTFS automaticamente cerca di comprimere i blocchi mentre sono scritti, ed automaticamente decomprime gli stessi quando vengono letti. I processi che leggono o scrivono file compressi sono completamente inconsapevoli del fatto che i meccanismi di compressione e decompressione sono attivi e vengono messi in atto.

La compressione di blocchi avviene come segue: quando NTFS scrive un file marcato per la compressione a disco, esamina i primi 16 blocchi (logici) nel file, senza badare a quante sequenze essi occupano. Quindi esegue un algoritmo di compressione su di questi, e se i dati risultanti possono essere immagazzinati in 15 blocchi o meno, i dati com-

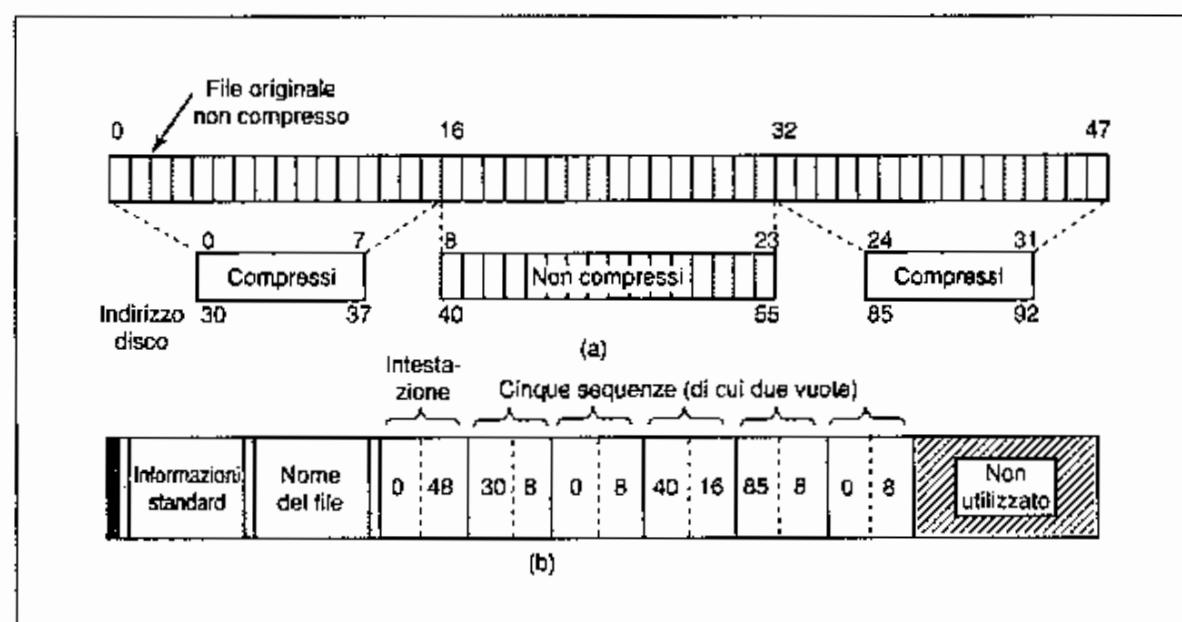


Figura 11.40 (a) Un esempio di un file a 48 blocchi compresso a 32 blocchi. (b) Il record MFT per il file, dopo la compressione.

pressi vengono scritti su disco, se possibile in una sequenza; se invece occupano comunque 16 blocchi, i 16 blocchi vengono scritti in forma non compressa. Successivamente vengono esaminati i blocchi 16-31 per vedere se possono essere compressi a 15 blocchi o meno, e così via.

La Figura 11.40(a) mostra un file nel quale i primi 16 blocchi sono stati compressi con successo ad otto blocchi, i secondi 16 blocchi hanno fallito la compressione, ed anche i terzi 16 blocchi sono stati compressi del 50%; le tre parti sono state scritte come tre sequenze ed immagazzinate nel record MFT. I blocchi "mancanti" sono stati immagazzinati nel record MFT all'indirizzo di disco 0, come mostrato in Figura 11.40 (b); l'intestazione (0,48) è seguita da cinque coppie, due per la prima sequenza (compressa), una per la seconda sequenza non compressa, e due per la sequenza (compressa) finale.

Quando il file viene letto, NTFS deve conoscere quali sequenze sono compresse e quali non lo sono: lo vede in base agli indirizzi disco, perché un indirizzo disco a 0 indica la parte finale di 16 blocchi compressi. Infatti il blocco disco 0 non può essere utilizzato per immagazzinare dati, per eliminare ambiguità, e perché contiene il settore di avvio; quindi il suo utilizzo per i dati è impossibile in ogni caso.

Un accesso casuale ad un file compresso è possibile, anche se laborioso. Si supponga che un processo faccia una richiesta per il blocco 35 in Figura 11.40; come fa NTFS a localizzare il blocco 35 in un file compresso? La risposta è che deve leggere e decomprimere prima l'intera sequenza, dopodiché conosce dove si trova il blocco 35 e lo può passare al processo che lo desidera leggere. La scelta di 16 blocchi per unità di compressione è stata un compromesso, rendendola più corta si sarebbe ottenuta una compressione meno efficace, mentre una scelta superiore a 16 avrebbe reso l'accesso casuale più costoso.

## Cifratura di file

Oggi i computer sono utilizzati per immagazzinare ogni tipo di dati sensibili, tra cui piani di acquisizione di aziende, informazioni sulla tasse e lettere d'amore (e-mail d'amore?), che i proprietari non vogliono rendere accessibili a nessuno. La perdita di informazioni può avvenire nel caso in cui un computer portatile viene perso o rubato, od un computer da tavolo viene avviato con un floppy disk MS-DOS saltando le sicurezze di Windows 2000, o ancora se un hard disk viene rimosso dal computer ed installato in un altro con un sistema operativo non sicuro. Anche semplicemente l'andare in bagno e lasciare il computer incustodito per qualche minuto, potrebbe comportare una rottura della sicurezza dei dati.

Windows 2000 affronta questi problemi avendo l'opzione di cifrare i file, in modo che, anche nel caso in cui il computer venga rubato o riavviato usando MS-DOS, i file cifrati rimangano inaccessibili. Il modo standard per usare la cifratura di Windows 2000 è quello di marcare certe directory come cifrate, il che causa la cifratura di tutti i file in esse contenuti, oltre che dei nuovi file spostati in esse o creati. La cifratura e la decifratura effettiva non vengono fatte da NTFS, ma da un driver chiamato EFS (Encrypting File System - File System di cifratura), che viene posto tra NTFS ed il programma utente. In questo modo, i programmi applicativi non sono consapevoli della cifratura, ed anche NTFS è solo parzialmente impegnato nel procedimento.

Per capire come lavora il file system di cifratura, è necessario capire come funziona la moderna crittografia; a questo proposito, una breve revisione è stata fornita nella Sezione 9.2, ed i lettori non familiari con questi concetti di base di crittografia dovrebbero rileggere la sezione, prima di continuare.

Adesso vediamo come Windows 2000 cifra i file: quando un utente chiede che un file venga cifrato, una chiave a 128 bit viene generata in modo casuale e quindi utilizzata per

cifrare il file blocco a blocco utilizzando un algoritmo simmetrico parametrizzato su questa chiave. Ogni nuovo file cifrato riceve una diversa chiave a 128 bit per file, cosicché non possono esistere due file che utilizzano la stessa chiave di cifratura, il che aumenta la sicurezza nel caso in cui una chiave venga compromessa. L'algoritmo di compressione attuale è una variante di DES (Data Encryption Standard - Standard di cifratura di dati), ma l'architettura EFS supporta l'aggiunta di nuovi algoritmi in futuro. La cifratura di blocchi di file in modo indipendente è indispensabile per poter permettere l'accesso casuale ai file.

La chiave del file deve essere immagazzinata da qualche parte in modo che il file possa essere decifrato in seguito; se questa viene salvata in chiaro da qualche parte nel computer, e questo viene rubato o perso, la chiave sarà facilmente recuperabile, compromettendo lo scopo stesso della crittografia. Per questa ragione, le chiavi dei file devono essere a loro volta cificate prima di essere immagazzinate nel disco; per questo scopo si usano gli approcci a chiave pubblica.

Dopo la cifratura dei file, la posizione della chiave pubblica dell'utente viene recuperata utilizzando le informazioni del registro; non vi è pericolo nel salvare la chiave pubblica nel registro, poiché anche se un ladro rubasse il computer e la recuperasse, non vi sarebbe modo di risalire alla chiave privata da essa. La chiave a 128 bit casuale per il file viene quindi cificata con la chiave pubblica, ed il risultato viene immagazzinato su disco insieme con il file, come mostrato in Figura 11.41.

Per decifrare il file, viene recuperata dal disco la chiave cificata di 128 bit casuale relativa, ma per decifrarla e recuperarla, l'utente deve fornire la chiave privata. Idealmente, essa dovrebbe essere salvata in una smart card, esterna al computer, ed inserita solo quando il file deve essere recuperato e quindi decifrato. Sebbene Windows 2000 supporti le smart card, non memorizza, per ora, la chiave privata direttamente in esse.

Invece, al primo istante in cui un utente cifra un file utilizzando EFS, Windows 2000 genera una coppia (chiave privata, chiave pubblica) e salva la chiave privata su disco, cificata utilizzando un algoritmo simmetrico di cifratura. La chiave utilizzata per l'algoritmo simmetrico è derivata o dalla login e password dell'utente, oppure dalla chiave immagazzinata nella smart card, se la login di smart card è attiva. In questo modo, EFS può decifrare la chiave privata nell'istante di accesso al sistema e mantenerla nello spazio degli indirizzi virtuale durante le operazioni normali; quindi può decifrare

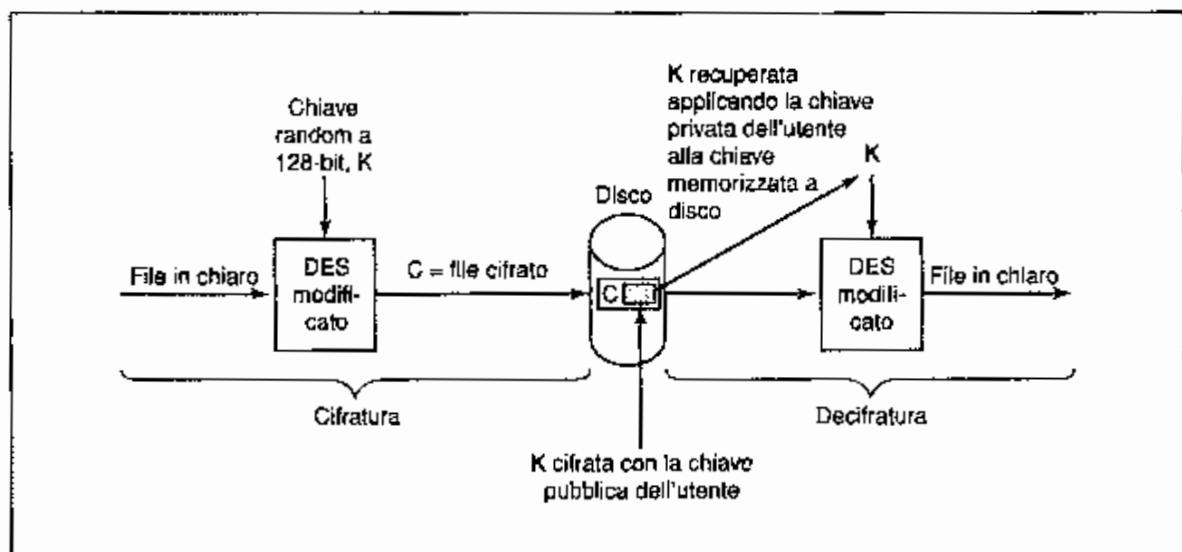


Figura 11.41 Come opera il file system di cifratura.

chiavi di file a 128 bit senza ulteriori operazioni di accesso a disco. Quando il computer viene spento, la chiave privata viene cancellata dallo spazio virtuale di indirizzi di EFS in modo che nessuno, rubando il computer, possa di nuovo avere accesso alla chiave privata del proprietario.

Possono esserci complicanze nel caso in cui molti utenti necessitano di un accesso allo stesso file cifrato; allo stato attuale delle cose la condivisione di file cifrati non è supportata. Ad ogni modo, l'architettura EFS potrebbe supportare tale condivisione nel futuro cifrando ogni file in modo multiple, ogni volta con la chiave pubblica relativa all'utente interessato; tutte queste versioni di file cifrati possono poi essere attaccate al file.

La necessità potenziale di condividere file cifrati è una delle ragioni per le quali viene utilizzando un sistema a due chiavi; se tutti i file sono stati cifrati dalla chiave del proprietario, non ci potrebbe essere un modo di condividerli, ma utilizzando chiavi diverse per cifrare ogni file, questo problema può essere risolto.

Avere una chiave random per file, cifrata con la chiave simmetrica del possessore non funziona poiché la chiave di cifratura simmetrica che viaggia in chiaro rovinerebbe la sicurezza - la generazione della chiave di decifratura dalla chiave di cifratura è troppo facile. Quindi è necessaria una (lenta) crittografia a chiave pubblica per cifrare le chiavi dei file, e poiché la chiave di cifratura è pubblica in ogni modo, il fatto di averla disponibile non è pericoloso.

L'altra ragione per la quale si usa il sistema a due chiavi sono le prestazioni: usare una crittografia a chiave pubblica per cifrare ogni file è un procedimento troppo lento, ed è molto più efficiente l'utilizzo della crittografia a chiave simmetrica per cifrare i dati, e della crittografia a chiave pubblica per cifrare le chiavi simmetriche dei file.

## 11.8 La sicurezza in Windows 2000

Avendo appena parlato della cifratura nel file system, è il momento buono per esaminare la sicurezza. NT è stato progettato per soddisfare i requisiti di sicurezza C2 del Dipartimento di Difesa americano (DoD 5200.28-STL), l'Orange Book, che abbiamo studiato nel capitolo 9. Questo standard richiede che i sistemi operativi posseggano certe proprietà in modo da essere classificati come abbastanza sicuri per il lavoro militare. Sebbene Windows 2000 non fu specificatamente progettato per soddisfare i requisiti C2, esso eredita molte proprietà di sicurezza da NT comprese:

1. Accesso sicuro con misure antispoofing.
2. Controlli di accesso discrezionali.
3. Controlli di accesso privilegiati.
4. Protezione dello spazio degli indirizzi per ogni processo.
5. Le nuove pagine devono essere azzerate prima di essere mappate.
6. Auditing di sicurezza.

Rivediamo questi elementi brevemente (fra l'altro, nessuno di questi è presente in Windows 98).

L'accesso sicuro vuol dire che l'amministratore di sistema può richiedere che tutti gli utenti abbiano una password per poter accedere al sistema; si ha spoofing quando un utente malevolo scrive un programma che mostra il prompt di accesso o lo schermo e successivamente si allontana dal computer nella speranza che un utente innocente si sie-

da ed inserisca nome e password. Il nome e la password sono scritti a disco e all'utente viene detto che l'accesso è fallito. Windows 2000 impedisce questo attacco obbligando gli utenti a premere CTRL-ALT-DEL per accedere: tale sequenza di tasti viene catturata sempre dal driver di tastiera, che chiama un programma di sistema che crea la vera finestra di accesso. Questa procedura funziona poiché non vi è modo per i processi utente di disattivare il controllo del driver di tastiera sulla sequenza CTRL-ALT-DEL.

I controlli di accesso discrezionali permettono al proprietario di un file o di un altro oggetto di comunicare chi lo può usare, ed in quale modo. I controlli di accesso privilegiati permettono che l'amministratore di sistera (superuser) li sovrascriva quando necessario; la protezione dello spazio degli indirizzi vuol dire solamente che ogni processo ha il proprio spazio degli indirizzi virtuale, non accessibile da nessun altro processo non autorizzato. Il successivo elemento indica che quando lo stack aumenta, le pagine mappate vengono inizializzate a 0, in modo che i processi non possono trovare vecchie informazioni inserite nelle pagine da precedenti proprietari (segue da ciò che la lista di pagine azzerate in Figura 11.27 costituisce un rifornimento di pagine azzerate per questo scopo). Infine l'auditing di sicurezza permette che l'amministratore produca un log di certi eventi relativi alla sicurezza.

Nella prossima sezione descriveremo i concetti di base dietro la sicurezza di Windows 2000, dopodiché daremo uno sguardo alle chiamate di sicurezza di sistema, ed infine concluderemo vedendo come la sicurezza viene implementata.

### 11.8.1 Concetti fondamentali

Ogni utente Windows 2000 (e gruppo) è identificato da un SID (Security ID, ID di sicurezza). I SID sono numeri binari con una breve intestazione seguita una lunga componente casuale; ogni SID deve essere unico al mondo, e quando un utente attiva un processo, il processo ed i suoi thread girano sotto il SID dell'utente. La maggior parte dei sistemi di sicurezza è progettata in modo che solo i thread con SID autorizzati possono accedere agli oggetti interessati.

Ogni processo ha un token di accesso che specifica il SID relativo ed altre proprietà; esso viene normalmente assegnato al momento dell'accesso al sistema da *winlogon* ed è mostrato in Figura 11.42, anche se sarebbe meglio che i processi chiamassero *GetTokenInformation* per acquisirlo, perché questo procedimento potrebbe cambiare in futuro. L'intestazione contiene alcune informazioni di tipo amministrativo; il campo con il tempo di scadenza comunica quando il token cessa di essere valido, ma attualmente non viene utilizzato; i campi Gruppi specificano i gruppi ai quali il processo appartiene; tutto ciò è necessario per compatibilità con POSIX. Il DACL (ACL discrezionale) è una lista di controllo di accesso assegnata ad oggetti creati dal processo quando nessun altro ACL viene specificato. Il SID dell'utente comunica chi possiede il processo. I SID ristretti esistono per permettere ai processi non controllati di prendere parte ad operazioni con processi sicuri, ma con minor potere di fare danno.

Infine i privilegi elencati, se esistenti, forniscono poteri speciali al processo, come ad esempio il potere di spegnere la macchina ed accedere a file che altrimenti non potrebbero essere acceduti. In effetti, i privilegi suddividono i poteri del superuser in parecchi diritti che possono essere assegnati individualmente ai processi. In questo modo un utente può acquisire alcuni diritti da superuser, ma non tutti; in sostanza, il token di accesso comunica chi possiede un processo e quali poteri per difetto sono associati ad esso.

Quando un utente accede al sistema, *winlogon* dà al processo iniziale un token di accesso, ed i processi successivi normalmente ereditano questo token; anche il token di

| Intestazione | Data di scadenza | Gruppi | CACL per difetto | SID utente | SID di gruppo | SID ristretti | Privilegi |
|--------------|------------------|--------|------------------|------------|---------------|---------------|-----------|
|              |                  |        |                  |            |               |               |           |

Figura 11.42 Struttura di un token di accesso.

accesso di un processo, inizialmente, viene ereditato da tutti i suoi thread. Ad ogni modo un thread può prendere un token di accesso differente durante l'esecuzione, nel qual caso tale token viene scritto sopra quello ereditato dal processo. In particolare, un thread client può passare il proprio token di accesso al thread server per permettere al server di accedere ai file protetti del client e ad altri oggetti. Questo meccanismo viene chiamato **impersonation** (personificazione).

Un altro concetto fondamentale è il **descrittore di sicurezza** (security descriptor). Ogni oggetto ha un descrittore di sicurezza associato che comunica chi può operare su di esso e quali operazioni può compiere. Un descrittore di sicurezza si compone di una intestazione, seguita da un DACL con uno o più ACE (elementi di controllo di accessi - Access Control Elements). I due principali tipi di elementi sono Allow e Deny (Permesso e Divieto); un elemento Allow specifica un SID ed una mappa di bit che specifica quali operazioni possono essere attuate sull'oggetto dai processi con quel SID. Un elemento Deny lavora nello stesso modo, salvo il fatto che quando trova la corrispondenza, ciò significa che il chiamante non può concludere l'operazione richiesta. Per esempio, Ida ha un file il cui descrittore di sicurezza specifica che tutti hanno accesso di lettura, Elvis non ha accesso, Cathy ha accesso di lettura e scrittura, e Ida ha accesso completo. Questo sem-

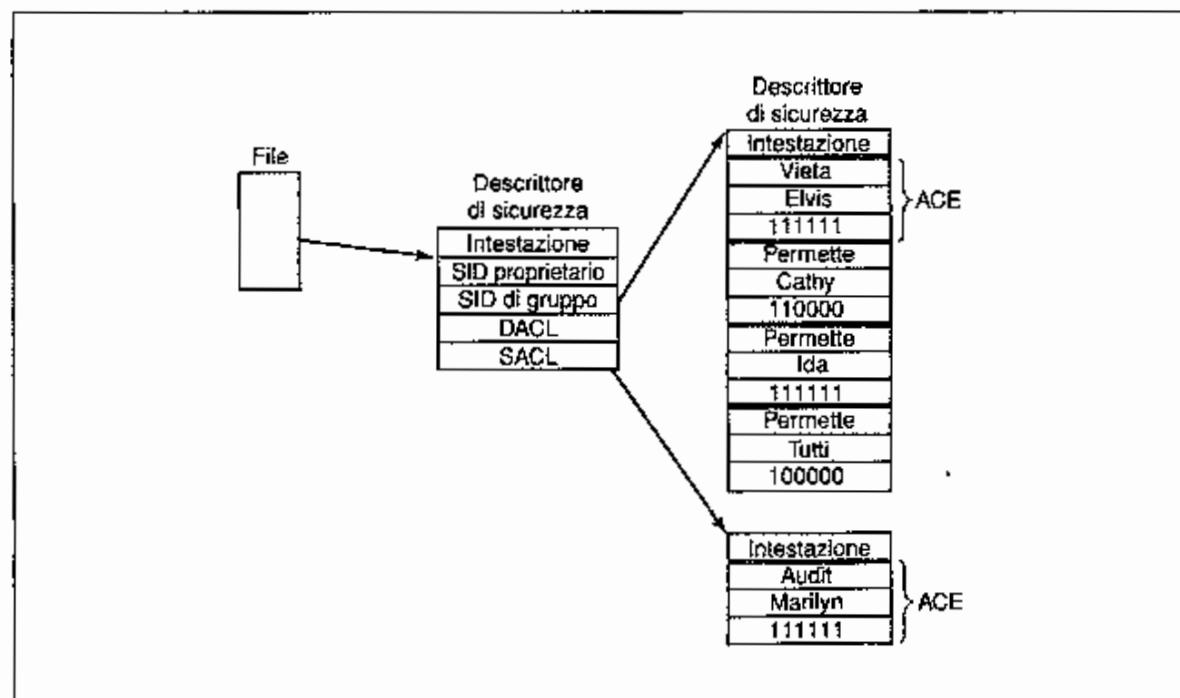


Figura 11.43 Un esempio di descrittore di sicurezza per un file.

plice esempio viene illustrato in Figura 11.43: il SID Tutti si riferisce all'insieme di tutti gli utenti, ma esso viene sovrascritto da un qualsiasi ACE che segue.

In aggiunta al DACL, un descrittore di sicurezza possiede anche un **SACL** (lista di controllo di accessi di sistema - System Access Control List), che è come una DACL che non specifica chi può utilizzare l'oggetto, ma quali operazioni sull'oggetto sono registrate nel log di eventi di sicurezza dell'intero sistema; in Figura 11.43, ogni operazione che Marilyn compie sul file viene registrata. Windows 2000 fornisce funzionalità di auditing addizionali per il log di accessi sensibili.

## 11.8.2 Chiamate API di sicurezza

La maggior parte dei meccanismi di controllo di accessi in Windows 2000 è basata sui descrittori di sicurezza, il percorso standard è che quando un processo crea un oggetto, esso fornisca anche un descrittore di sicurezza come parametro di `CreateProcess`, `CreateFile` o altre chiamate di creazione di oggetti. Questo descrittore di sicurezza quindi diviene il descrittore di sicurezza attaccato all'oggetto, come abbiamo visto in Figura 11.43. Se non viene fornito nessun descrittore di sicurezza nella chiamata di creazione dell'oggetto, si usa la sicurezza per difetto del token di accesso del chiamante (si veda Figura 11.42).

Molte delle chiamate di sicurezza API Win32 sono legate alla gestione del descrittore di sicurezza, quindi noi ci soffermeremo su di esse; le più importanti sono elencate in Figura 11.44. Per creare un descrittore di sicurezza, prima si alloca lo spazio, e quindi lo si inizializza usando `InitializeSecurityDescriptor`; questa chiamata ne riempie l'intestazione. Se il SID del possessore non è noto, può essere cercato per nome, utilizzando `LookupAccountSid`, e successivamente questo può essere inserito nel descrittore di sicurezza. Lo stesso avviene per il SID di gruppo, se esiste. Normalmente questi saranno i SID del chiamante e di uno dei suoi gruppi, ma l'amministratore di sistema può utilizzare qualsiasi SID.

A questo punto la DACL (o SACL) del descrittore di sicurezza può essere inizializzata con `InitializeAcl`; i campi dato ACL possono essere aggiunti chiamando `AddAccessAllowedAce`, e `AddAccessDeniedAce`, funzioni che possono essere usate più volte per aggiungere tanti campi ACE quanto necessario. `DeleteAce`, può essere utilizzata per rimuovere un campo dato, quindi per una ACL esistente piuttosto che per una appena costruita. Quando un ACL è pronto, si può usare `SetSecurityDescriptor-`

| Funzione API Win32                        | Descrizione                                                    |
|-------------------------------------------|----------------------------------------------------------------|
| <code>InitializeSecurityDescriptor</code> | Prepara un nuovo descrittore di sicurezza                      |
| <code>LookupAccountSid</code>             | Recupera il SID per un certo nome utente                       |
| <code>SetSecurityDescriptorOwner</code>   | Inserisce il SID del proprietario nel descrittore di sicurezza |
| <code>SetSecurityDescriptorGroup</code>   | Inserisce il SID del gruppo nel descrittore di sicurezza       |
| <code>InitializeAcl</code>                | Inizializza DACL o SACL                                        |
| <code>AddAccessAllowedAce</code>          | Inserisce un nuovo ACE in DACL (SACL) permettendo l'accesso    |
| <code>AddAccessDeniedAce</code>           | Inserisce un nuovo ACE in DACL (SACL) vietando l'accesso       |
| <code>DeleteAce</code>                    | Rimuove un ACE da DACL (SACL)                                  |
| <code>SetSecurityDescriptorDacl</code>    | Attacca un DACL ad un descrittore di sicurezza                 |

Figura 11.44 Le principali funzioni API Win32 per la sicurezza.

DACL per attaccarlo al descrittore di sicurezza. Infine, quando un oggetto viene creato, si può passare come parametro il nuovo descrittore di sicurezza, perché venga attaccato all'oggetto.

### 11.8.3 Implementazione della sicurezza

La sicurezza in un sistema Windows 2000 stand-alone è implementata utilizzando un certo numero di componenti che abbiamo già visto (la sicurezza nelle reti è una storia a parte che va al di là degli scopi di questo libro); l'accesso al sistema è gestito da *winlogon* e l'autenticazione da *lsass* e *msasn1.dll* come discusso in Sezione 11.4.5. Il risultato di un accesso con successo è una nuova shell con il relativo token di accesso associato. Questo processo usa le chiavi SECURITY e SAM nel registro; la prima definisce le politiche di sicurezza generale, e la seconda contiene le informazioni di sicurezza per gli utenti individuali come discusso in Sezione 11.2.3.

Una volta che un utente è entrato, le operazioni di sicurezza avvengono quando apre un oggetto nel tentativo di accedervi. Ogni chiamata *OpenXXX* richiede il nome dell'oggetto da aprire e l'insieme di diritti necessari; durante il processo di apertura, il gestore di sicurezza (vedi Figura 11.7) controlla per vedere se il chiamante ha tutti i diritti richiesti, ed attua questo controllo guardando il token di accesso del chiamante e la DACL associata all'oggetto. Esso procede nella lista di campi dati nell'ACL in sequenza, e non appena trova un campo dato che coincide con il SID del chiamante o con uno dei gruppi del chiamante, l'accesso trovato è preso come definitivo. Se tutti i diritti che il chiamante richiede sono disponibili, l'apertura dell'oggetto ha successo; altrimenti fallisce.

Le DACL possono avere campi dato Deny e Allow, come abbiamo già visto; per questa ragione, di solito si inseriscono nella ACL i campi dato che negano l'accesso prima dei campi dato che permettono l'accesso, cosicché un utente che ha specificatamente negato l'accesso non lo può ottenere per via indiretta essendo membro di un gruppo a cui l'accesso è permesso.

Dopo che un oggetto è stato aperto si restituisce al chiamante un gestore, e nelle chiamate successive, l'unico controllo che viene fatto è se l'operazione richiesta al momento appartiene all'insieme di operazioni decise in fase di apertura, per impedire che un chiamante che aveva aperto un file in lettura possa scriverci sopra. I campi dato di tipo log richiesti da una SACL sono sempre creati.

## 11.9 Caching in Windows 2000

Il gestore di cache di Windows 2000 opera il caching per ragioni di prestazioni, in modo simile concettualmente alle cache in altri sistemi operativi. Ad ogni modo, il suo progetto possiede alcune proprietà insolite che vale la pena osservare brevemente.

Il lavoro del gestore di cache è di mantenere in memoria i blocchi di file system che sono stati usati recentemente, per ridurre i tempi di accesso in caso di riferimenti successivi. Windows 2000 ha una sola cache integrata che funziona per tutti i file system in uso, tra cui NTFS, FAT-32, FAT-16 e anche il file system del CD-ROM; ciò significa che i file system non hanno bisogno di mantenere le proprie cache.

Come conseguenza dello scopo di progettare una singola cache integrata, anche in presenza di molti file system, il gestore di cache viene posto in una strana posizione nel sistema, come abbiamo visto in Figura 11.7: esso non è parte di un file system poiché vi sono file system indipendenti che possono non avere nulla in comune. Di contro, esso opera ad un livello più alto dei file system, che sono tecnicamente driver sotto il control-

lo del gestore di I/O.

La cache di Windows 2000 è organizzata in blocchi virtuali, non blocchi fisici: per capire cosa vuol dire ricordiamoci che le cache tradizionali di file tengono traccia di blocchi mediante l'utilizzo di indirizzi a due parti, della forma (partizione, blocco), dove il primo membro denota il dispositivo e la partizione, ed il secondo membro è un numero di blocco nella partizione. Il gestore di cache di Windows 2000 non lavora così: esso usa la copia (file, offset) per riferirsi ad un blocco (teoricamente la cache viene utilizzata per i flussi e non per i file, ma noi ignoreremo questo dettaglio).

La ragione di questa soluzione non ortodossa è che quando arriva una richiesta al gestore di cache, essa è specificata come (file, offset) poiché questo è tutto ciò che il processo chiamante conosce. Se i blocchi di cache fossero etichettati con un marcitore di tipo (partizione, blocco), il gestore di cache non avrebbe modo di conoscere quale blocco (file, offset) corrisponde a quale blocco (partizione, blocco), poiché è il file system che mantiene questa mappa.

Esaminiamo adesso come opera il gestore di cache: quando un file viene riferito, il gestore di cache mappa un frammento di 256 KB dello spazio di indirizzi virtuale del kernel sul file; se il file è più grande di 256 KB, si mappa solo una porzione del file. Il quantitativo totale di spazio degli indirizzi virtuale che il gestore di cache può utilizzare è determinato in fase di avvio, e dipende dal totale di RAM presente; se il gestore di cache fuoriesce dai 256 KB dello spazio degli indirizzi virtuale, esso deve eliminare la mappa di un vecchio file per inapparne uno nuovo.

Una volta mappato un file, il gestore di cache può soddisfare le richieste dei blocchi semplicemente copiando dallo spazio degli indirizzi virtuale del kernel al buffer dell'utente; se il blocco copiato non è nella memoria fisica, avverrà un fault di pagina che il gestore della memoria soddisferà nel modo usuale. Al gestore di cache non interessa se il blocco era nella cache o meno, e l'operazione di copiatura ha sempre successo.

L'operato del gestore di cache è mostrato in Figura 11.45, nel caso di un file system NTFS su un disco SCSI e di un file system FAT-32 su disco IDE. Quando un processo opera una lettura di un file la richiesta viene passata al gestore di cache; se il blocco necessario è nella cache, esso viene copiato all'utente immediatamente, mentre se non è nella cache il gestore di cache ottiene un fault di pagina quando prova a copiarlo. Una volta servito il fault di pagina, il blocco viene copiato per il processo chiamante.

Come conseguenza di questo progetto, il gestore di cache non conosce quante pagine da lui mappate sono in memoria fisica o quanto è larga la sua cache; solo il gestore della memoria conosce queste funzionalità. Questo approccio fa sì che il gestore della memoria gestisca dinamicamente la dimensione della cache insieme con la memoria fisica per le pagine utente: se vi è poca attività di file ma vi sono molti processi attivi, il gestore di memoria può utilizzare la maggior parte della memoria fisica per le pagine dei processi; viceversa, se vi è molta attività di file e pochi processi, la memoria fisica può essere destinata alla cache.

Un'altra proprietà che il gestore di cache possiede è il mantenere coerenza tra i file mappati in memoria ed i file che sono aperti in lettura e scrittura: si consideri, per esempio, una situazione in cui un processo apre un file in lettura e scrittura, ed un secondo processo mappa questo file nel proprio spazio di indirizzi. Cosa succede se il secondo processo scrive sul file direttamente, e subito dopo il primo legge il blocco appena cambiato? Recupera dati scaduti?

La risposta è no: in entrambi i casi - file aperti e file mappati - il gestore di cache mappa un pezzo di 256 KB del suo spazio degli indirizzi virtuale sul file, ed il file viene mappato una sola volta, non importa quanti processi lo possiedono aperto o mappato. Nel caso di file mappato, entrambi (il gestore di cache ed il processo utente) condividono le pagine in memoria, e quando si deve soddisfare una richiesta di lettura, il gestore di

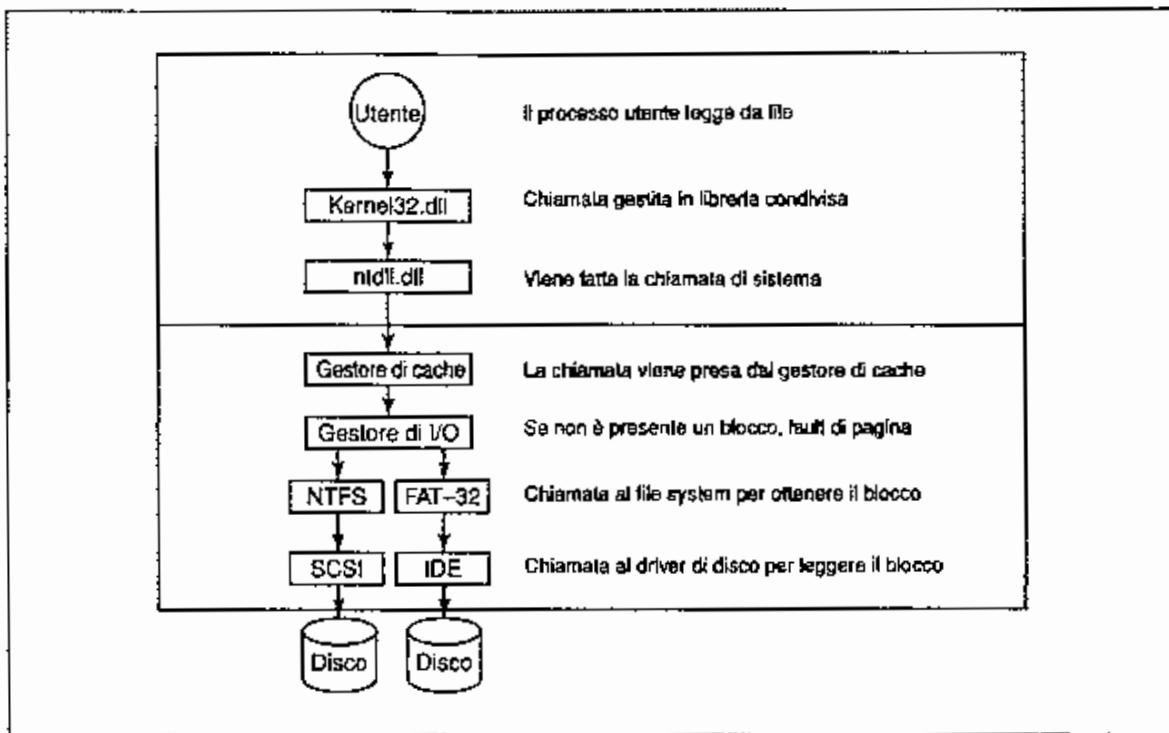


Figura 11.45 Il cammino dalla cache all'hardware.

cache copia semplicemente una pagina dalla memoria al buffer dell'utente, che così riflette sempre lo stato corrente del file, perché il gestore di cache sta utilizzando le stesse pagine del processo che ha mappato il file.

## 11.10 Sommario

Windows 2000 è composto dalla componente HAL, il Kernel, l'esecutivo, ed un sottile strato di servizi di sistema che recuperano le chiamate di sistema in arrivo; in più, c'è una varietà di driver di dispositivo, tra cui i file system ed il GDI. HAL nasconde certe differenze dell'hardware agli strati superiori, ed il kernel cerca di nascondere all'esecutivo le differenze rimanenti, in modo da rendere quest'ultimo completamente indipendente dalla macchina.

L'esecutivo è basato su oggetti in memoria, che i processi utenti possono creare, ottenendo dei gestori per manipolarli in seguito; anche le componenti dell'esecutivo possono creare oggetti. Il gestore degli oggetti mantiene uno spazio dei nomi, nel quale gli oggetti possono essere inseriti e successivamente cercati.

Windows 2000 supporta processi, job, thread e thread leggeri. I processi hanno spazi di indirizzi virtuali e sono contenitori di risorse; i thread sono unità di esecuzione e vengono schedulati dal sistema operativo, i thread leggeri, o fiber, sono thread più leggeri che sono schedulati interamente nello spazio utente; i job sono collezioni di processi, usati per l'assegnazione di quote di risorse. Lo scheduling viene fatto utilizzando un algoritmo di priorità nel quale si esegue successivamente il thread pronto con il più alto valore di priorità.

Windows 2000 supporta una memoria virtuale paginata a richiesta; l'algoritmo di paginazione è basato sul concetto di insieme di lavoro. Il sistema mantiene parecchie liste

di pagine libere, in modo che quando avviene un fault di pagina, vi è quasi sempre una pagina libera disponibile. Le liste di pagine libere sono alimentate mediante formule complesse sugli insiemi di lavoro, formule che cercano di buttare le pagine che non sono state usate per lungo tempo.

L'I/O viene fatto dai driver di dispositivo che seguono il modello di dispositivo di Windows; ogni driver incomincia il proprio lavoro inizializzando un oggetto driver, che contiene gli indirizzi delle procedure che il sistema può chiamare per aggiungere dispositivi o attuare operazioni di I/O. I driver possono essere impilati per agire come filtri.

Il file system NTFS è basato su una tabella principale di file, che possiede un record per ogni file o directory; ogni file possiede molti attributi, che possono essere o presenti nel record MFT o non residenti, cioè sul disco. NTFS supporta la compressione e la cifratura di file, tra le altre cose.

La sicurezza è basata sulle liste di controllo di accessi: ogni processo ha un token di controllo di accesso che comunica chi è e se possiede privilegi speciali; ogni oggetto ha un descrittore di sicurezza ad esso associato. Il descrittore di sicurezza punta ad una lista di controllo di accessi discrezionale, che contiene i campi di controllo di accesso, che possono permettere o impedire l'accesso ad individui o gruppi.

Infine, Windows 2000 mantiene una sola cache per tutti i file system, che è una cache virtuale piuttosto che una cache fisica. Le richieste per i blocchi disco passano prima per la cache, e se non possono essere soddisfatte si chiama il file system appropriato per recuperare i blocchi necessari.

## PROBLEMI

1. Formire un vantaggio e uno svantaggio dell'utilizzo del file di registry rispetto ad avere singoli file .ini
2. Un mouse può avere 1, 2, o 3 pulsanti; si usano normalmente tutti e tre i tipi. HAL nasconde questa differenza al resto del sistema operativo? Perché, ed in caso contrario, perché no?
3. HAL tiene traccia della partenza del timer nell'anno 1601. Dare un esempio di un'applicazione dove questa caratteristica risulta utile.
4. Il sottosistema POSIX deve implementare segnali in stile UNIX. Se un utente preme il tasto per il segnale di Quit, questo viene schedulato come un DPC o come un APC?
5. Molti componenti dell'esecutivo (Figura 11.7) chiamano altre componenti dell'esecutivo. Dare tre esempi di componente che ne chiama un altro, utilizzando sei differenti componenti in tutto.
6. Win32 non ha i segnali. Se fossero stati introdotti, sarebbero potuti essere per processo, per thread, per entrambi o per nessuno. Fare una proposta e spiegare perché è una buona idea.
7. Un'alternativa all'utilizzo delle DLL è linkare staticamente ogni programma con esattamente quelle procedure di libreria che effettivamente esso chiama, né più e né meno. Se questo schema fosse stato introdotto, avrebbe avuto più senso nelle macchine client o nelle macchine server?
8. Il file *ntdll.dll* esporta 1179 chiamate di funzione mentre *ntoskrnl.exe* esporta 1209 chiamate di funzione. È un baco? Quale potrebbe essere la causa di questa discrepanza?

9. Gli oggetti gestiti dal gestore di oggetti sono di dimensioni variabili, con differenti oggetti aventi differenti dimensioni. Può un oggetto partire ad un byte arbitrario nel pool non paginato? Suggerimento: Non è necessaria nessuna informazione su Windows 2000 oltre a quelle fornite nel testo.
10. C'è qualche limite al numero di differenti operazioni che possono essere definite in un oggetto esecutivo? Se sì, da dove derivano questi limiti? Se no, perché no?
11. La chiamata API Win32 `WaitForMultipleObjects` permette ad un thread di bloccarsi su un insieme di oggetti di sincronizzazione, i gestori dei quali sono passati come parametri. Non appena uno di essi riceve un segnale, il thread chiamante è rilasciato. È possibile avere un insieme di sincronizzazione contenente due semafori, un mutex, e una sessione critica? Perché si o perché no? Suggerimento: questo non è un trabocchetto ma richiede un po' di attento ragionamento.
12. Nominare tre ragioni per le quali un processo può essere fatto terminare.
13. Considerare la situazione di Figura 11.19 nella quale il sistema sta per schedulare un thread. Supponendo che ogni thread sia basato su calcoli intensivi, quanto ci vuole prima che i thread a priorità 3 ottengano di essere eseguiti, su Windows 2000 Professional?
14. Supponiamo che il quanto sia definito a 20 ms e il thread corrente, a priorità 24, abbia appena cominciato il quanto. Improvvisamente termina una operazione di I/O, e un thread a priorità 28 diventa pronto. Quanto tempo deve aspettare quest'ultimo, all'incirca, prima di essere servito?
15. In Windows 2000, la priorità corrente è sempre maggiore o uguale alla priorità di base. Esistono circostanze per le quali avrebbe senso avere la priorità corrente minore di quella di base? Se sì, dare un esempio. Se no, perché?
16. Alcuni programmi MS-DOS furono scritti in linguaggio assembler utilizzando istruzioni come `IN port` ed `OUT port`. Possono questi programmi funzionare sotto Windows 2000? Se no, si può pensare ad un modo con il quale possano essere supportati?
17. Nominare due modi per offrire miglior tempo di risposta a processi importanti.
18. Il codice indipendente dalla posizione è già stato brevemente discusso in questo testo: è una tecnica per permettere a due processi di condividere la stessa procedura a differenti indirizzi virtuali. Si può risolvere il problema impostando la tabella delle pagine dei due processi in modo appropriato? Spiegare la risposta.
19. Le librerie condivise in Windows 2000 sono contenute in file `.dll` che più processi possono mappare contemporaneamente. Se due processi necessitano di mappare la stessa libreria condivisa a differenti indirizzi virtuali, c'è un problema. Come lo si può risolvere nei Pentium, utilizzando una proprietà della loro architettura di memoria? Se la soluzione richiede cambiamenti all'implementazione di Windows 2000, stabilire quali cambiamenti sono necessari.
20. Se un'area dello spazio degli indirizzi virtuali è riservata ma non impegnata, pensi venga creato un VAD per questa? Difendi la tua risposta.
21. Quali fra le transizioni mostrate in Figura 11.27 sono decisioni politiche, in contrapposizione a mosse forzate da eventi di sistema (es. un processo esce e libera le sue pagine)?

22. Supponiamo che una pagina sia condivisa, e contemporaneamente presente in due insiemi di lavoro; se essa è eliminata da uno degli insiemi di lavoro, dove va a finire in Figura 11.27?
23. Quando un processo toglie dalla mappatura una pagina vuota, esso fa la transizione (5) di Figura 11.27. Perché non ci sono transizioni alla lista modificata quando una pagina sporca viene tolta dalla mappatura?
24. Supponiamo che sia la RAM sia la video RAM possano entrambe leggere o scrivere una parola a 32 bit in 10 ms, quanto tempo ci vuole per riempire il background di una schermata XGA nel migliore dei casi?
25. Un file ha la seguente mappatura. Fornire le sequenze di campo dato MFT.
- | Offset          | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9 | 10 |
|-----------------|----|----|----|----|----|----|----|----|----|---|----|
| Indirizzo disco | 50 | 51 | 52 | 22 | 24 | 25 | 26 | 53 | 54 | - | 60 |
26. Considerare il record MFT di Figura 11.36; supponiamo che il file cresca, rendendo necessario un decimo blocco alla fine del file; il numero di questo blocco è 66. Come appare il record MFT ora?
27. In Figura 11.40(b), le prime due sequenze sono ognuna 8 blocchi di lunghezza. È incidentale che siano uguali o ha a che fare con la modalità di compressione? Spiegare la risposta.
28. Supponi di voler costruire Window 2000 Lite. Quali dei campi di Figura 11.42 potrebbe essere rimosso senza indebolire la sicurezza del sistema?
29. Il comando *regedit* può essere utilizzato sotto tutte le correnti versioni di Windows per esportare parte o tutto il registro su un file di testo. Salva il registro diverse volte durante una sessione di lavoro e osserva cosa cambia. Se hai accesso ad un calcolatore Windows nel quale puoi installare software o hardware, evidenzia quali cambiamenti avvengono quando un dispositivo o un programma vengono aggiunti o rimossi.
30. Scrivere un programma UNIX che simula di scrivere un file NTFS con sequenze multiple. Esso dovrebbe accettare una lista di uno o più file come argomenti, e scrivere un file di output che contenga una sequenza con gli attributi di tutti gli argomenti, e sequenze addizionali con il contenuto di ognuno degli argomenti. Scrivere poi un secondo programma per chiedere informazioni sugli attributi e le sequenze, e per estrarre tutti i componenti.
31. Scrivere un programma che generi un nome di file MS-DOS 8+3 dato un qualsiasi nome file. Utilizzare l'algoritmo di Windows 2000.

# PROGETTAZIONE DI UN SISTEMA OPERATIVO

Negli undici capitoli precedenti, abbiamo trattato vari argomenti e dato un'occhiata a molti concetti ed esempi relativi ai sistemi operativi. Tuttavia, studiare un sistema operativo esistente è differente dal progettarne uno nuovo, ed in questo capitolo daremo una rapida occhiata ad alcuni problemi e compromessi che i progettisti dei sistemi operativi devono prendere in considerazione durante la progettazione e l'implementazione di un nuovo sistema.

Vi è una serie di tradizioni riguardo a ciò che è valido e ciò che non è valido nella progettazione di un sistema operativo, ma sorprendentemente è stato scritto poco a riguardo. Probabilmente il libro più importante è il classico libro di Fred Brooks (1975) *The Mythical Man Month* in cui riferisce le sue esperienze di progettazione e di implementazione del sistema operativo OS/360 dell'IBM; l'edizione del ventesimo anniversario rivede parte di questo materiale ed aggiunge quattro capitoli nuovi (Brooks, 1995). Probabilmente l'unico libro di testo sui sistemi operativi che si occupa seriamente della progettazione è *Operating Systems: A Design-Oriented Approach* (Crowley, 1997).

Tre articoli classici sulla progettazione di un sistema operativo sono "Hints for Computer System Design" (Lampson, 1984), "On Building Systems that Will Fail" (Corbató, 1991) e "End-to-End Arguments in System Design" (Saltzer ed altri, 1984). Come il libro di Brooks, tutti e tre gli articoli sono sopravvissuti negli anni estremamente bene; molti dei loro concetti sono validi ancora oggi come lo erano quando furono pubblicati per la prima volta.

Questo capitolo si basa su queste fonti, aggiungendo l'esperienza personale dell'autore come progettista e co-progettista di tre sistemi: Amoeba (Tanenbaum ed altri, 1990), MINIX (Tanenbaum e Woodhull, 1997), e Globe (Van Steen ed altri, 1999). Poiché non esis-

ste nessun accordo tra i progettisti di sistemi operativi sul modo migliore per progettare un sistema operativo, questo capitolo sarà più personale, speculativo e indubbiamente più controverso dei precedenti.

## 12.1 La natura del problema di progettazione

La progettazione di un sistema operativo è più un progetto ingegneristico che una scienza esatta, essendo difficile stabilire obiettivi chiari, e raggiungerli. Cominciamo a vedere questi concetti.

### 12.1.1 Obiettivi

Per progettare un sistema operativo di successo, i progettisti devono avere un'idea chiara di cosa vogliono; la mancanza di un obiettivo rende molto difficile prendere decisioni in seguito. Per chiarire questo punto è istruttivo dare un'occhiata ai due linguaggi di programmazione, PL/I e C. Il PL/I è stato progettato dall'IBM negli anni 60 in quanto era faticoso dover supportare sia il FORTRAN sia il COBOL, ed era imbarazzante sentire le chiacchiere accademiche di corridoio, che affermavano che l'Algol era meglio di entrambi. Così fondarono un comitato per produrre un linguaggio che fosse adatto per tutte le persone: PL/I. Esso aveva un po' di FORTRAN, un po' di COBOL e un po' d'Algol; ma il linguaggio è fallito in quanto mancava di una visione uniforme, ed era semplicemente una collezione di caratteristiche in contrasto l'una con l'altra e per giunta troppo ingombranti per essere compilate in maniera efficiente.

Ora consideriamo il C: è stato progettato da una persona (Dennis Ritchie) per uno scopo (la programmazione di sistema). Fu un enorme successo, poiché Ritchie sapeva cosa voleva e cosa non voleva, e come risultato, è ancora in largo uso decenni dopo la sua comparsa: avere una chiara visione di cosa si vuole è cruciale.

Cosa vogliono i progettisti di sistemi operativi? Questo, ovviamente, varia da sistema a sistema, sarà differente per sistemi embedded piuttosto che per sistemi server; tuttavia, per i sistemi operativi general-purpose vengono in mente 4 elementi principali:

1. Definire le astrazioni.
2. Fornire le operazioni primitive.
3. Assicurare l'isolamento.
4. Gestire l'hardware.

Ognuno di questi punti sarà discusso nel seguito.

Il compito più importante, ma probabilmente più difficile, di un sistema operativo è quello di definire le astrazioni giuste: alcune di loro, come ad esempio i processi ed i file, sono stati utilizzati così a lungo che sembrano ovvi, altri, come ad esempio i thread, sono più nuovi e meno maturi. Per esempio, se un processo multithread che ha un thread bloccato in attesa di un input da tastiera esegue una `fork`, anche nel nuovo processo c'è un thread in attesa di un input da tastiera? Altre astrazioni riguardano la sincronizzazione, i segnali, il modello della memoria, la gestione dell'I/O e molti altri campi.

Ogni astrazione può essere istanziata come struttura dati concreta: gli utenti possono creare processi, file, semafori eccetera. Le operazioni primitive manipolano queste strutture dati: per esempio, gli utenti possono leggere e scrivere file e tali operazioni primiti-

ve sono implementate nella forma di chiamate di sistema. Dal punto di vista dell'utente, il cuore di un sistema operativo è composto dalle astrazioni, e dalle operazioni disponibili su di esse tramite le chiamate di sistema.

Poiché diversi utenti possono essere collegati contemporaneamente ad uno stesso computer, il sistema operativo ha la necessità di fornire meccanismi per tenerli separati: un utente non può interferire con un altro utente. Il concetto di processo è largamente usato per raggruppare insieme risorse per scopi di protezione, e generalmente, anche i file e le altre strutture dati sono protette. Assicurare ad ogni utente di poter eseguire solo le operazioni autorizzate sui dati autorizzati è un obiettivo chiave della progettazione di un sistema; tuttavia, gli utenti vogliono anche condividere dati e risorse, quindi l'isolamento deve essere selettivo e sotto il controllo dell'utente, e questo rende tutto più complicato.

Strettamente collegata a questo punto è la necessità di isolare i fallimenti: se qualche parte del sistema fallisce, più comunemente un processo utente, questa non dovrebbe essere in grado di far fallire con sé il resto del sistema. La progettazione di un sistema dovrebbe garantire che le varie parti siano ben isolate le une dalle altre; idealmente, le parti del sistema operativo dovrebbero anche essere isolate le une dalle altre per consentire fallimenti indipendenti.

Infine, il sistema operativo deve gestire l'hardware. In particolare, deve prendersi cura di tutti i chip a basso livello, come i controllori delle interruzioni e i controllori dei bus. Deve inoltre fornire una struttura per consentire ai driver dei dispositivi di gestire la maggior parte dei dispositivi di I/O, come ad esempio dischi, stampanti e video.

## 12.1.2 Perché è complicato progettare un sistema operativo?

La legge di Moore afferma che l'hardware di un computer migliora di un fattore 100 ogni dieci anni, ma nessuno ha una legge che afferma che i sistemi operativi migliorano di un fattore 100 ogni dieci anni, o addirittura che migliorano. Infatti, si potrebbe affermare in una causa che alcuni di loro sono peggiori, rispetto a fattori chiave come ad esempio l'affidabilità, della Versione 7 di UNIX degli anni '70.

Perché? L'inerzia ed il desiderio di una compatibilità a ritroso rispetto alle versioni precedenti spesso hanno la responsabilità maggiore, un'imputata è anche l'incapacità di aderire a buoni principi di progettazione. Ma c'è dell'altro: i sistemi operativi sono, per certi aspetti, fondamentalmente differenti dai piccoli programmi applicativi venduti nei negozi di computer per 49 dollari. Diamo un'occhiata a otto dei motivi che rendono la progettazione di un sistema operativo più difficile della progettazione di un programma applicativo.

Primo, i sistemi operativi sono diventati programmi estremamente grandi: nessuna persona può sedersi davanti ad un PC e improvvisare un sistema operativo importante in pochi mesi. Tutte le versioni correnti di UNIX superano il milione di linee di codice; Windows 2000 è 29 milioni di linee di codice e nessuna persona può comprendere 1 milione di linee di codice, figuriamoci 29 milioni di linee di codice. Quando si ha un prodotto che nessuno dei progettisti può sperare di comprendere completamente, non dovrebbe sorprendere che i risultati siano spesso distanti da quelli ottimali.

I sistemi operativi non sono i sistemi più complessi esistenti: le portaerei, ad esempio, sono molto più complicate, ma si suddividono meglio in sottosistemi isolati; le persone che progettano i servizi igienici su una portaerei non devono preoccuparsi del sistema radar ed i due sottosistemi non interagiscono molto. In un sistema operativo, il file system spesso interagisce con il sistema di menuaria in modi inaspettati ed imprevisti.

Secondo, i sistemi operativi devono occuparsi della concorrenza: ci sono diversi utenti e diversi dispositivi di I/O tutti attivi allo stesso tempo. Gestire la concorrenza è intrinsecamente molto più complicato che gestire una singola attività sequenziale, e le corse critiche (race condition) e lo stallo (deadlock) sono solo due dei problemi che possono presentarsi.

Terzo, i sistemi operativi devono occuparsi di utenti potenzialmente ostili - utenti che vogliono interferire con le operazioni di sistema o effettuare cose che sono proibite, come ad esempio rubare file di un altro utente. Il sistema operativo deve prendere provvedimenti per impedire a questi utenti di comportarsi in modo improprio; per contro i programmi di videoscrittura e i programmi di visualizzazione di fotografie non hanno questi problemi.

Quarto, a differenza del fatto che non tutti gli utenti si fidano l'uno dell'altro, molti utenti vogliono condividere alcune delle loro informazioni e delle loro risorse con altri utenti selezionati ed il sistema operativo deve rendere possibile ciò, ma in modo tale che gli utenti malvagi non possano interferire: anche in questo caso, i programmi applicativi non affrontano nessuna sfida simile a questa.

Quinto, i sistemi operativi vivono per un lungo periodo di tempo. UNIX è in circolazione da un quarto di secolo; Windows è in circolazione da oltre una decina d'anni e non mostra segni di scomparsa. Di conseguenza, i progettisti devono pensare a come l'hardware e le applicazioni potranno cambiare in un lontano futuro e dovrebbero prepararsi a questo cambiamento: i sistemi che sono troppo chiusi in una particolare visione del mondo generalmente muoiono uno ad uno.

Sesto, i progettisti dei sistemi operativi non hanno veramente un'idea valida di come i loro sistemi verranno utilizzati, quindi hanno la necessità di prevedere una considerevole generalità di uso: né UNIX né Windows furono progettati pensando alla posta elettronica o ai browser Web, eppure ora molti computer, utilizzando questi sistemi, fanno ben poco in più. Nessuno dice ad un progettista di navi di costruire una nave senza specificare se si vuole una nave da pesca, una nave da crociera o una nave da guerra, e ancora meno persone cambieranno la loro idea dopo che il prodotto è arrivato.

Settimo, i sistemi operativi moderni sono generalmente progettati per essere portabili, il che significa che devono girare su piattaforme hardware differenti, inoltre devono supportare centinaia, persino migliaia di dispositivi di I/O, tutti progettati in maniera indipendente, senza considerarsi l'un l'altro. Un esempio di dove questa diversità causa problemi è la necessità di avere un sistema operativo che gira sia su macchine little-endian sia su macchine big-endian; un secondo esempio si riscontrava costantemente sotto MS-DOS quando gli utenti volevano installare, ad esempio, una scheda audio e un modem che utilizzavano la stessa porta di I/O o le stesse linee di interruzione. Pochi programmi oltre ai sistemi operativi devono occuparsi dei problemi di ordinamento causati da elementi hardware in conflitto.

Ottavo, ed ultimo nella nostra lista, è la frequente necessità di essere compatibili a ritroso con qualche sistema operativo precedente: un tale sistema può avere restrizioni sulla lunghezza delle parole, sui nomi dei file, o su altri aspetti che i progettisti ora considerano obsoleti, ma a cui sono vincolati. È come convertire una fabbrica a produrre le macchine del prossimo anno invece che le macchine di quest'anno, mentre si continua a produrre le macchine di quest'anno a pieno ritmo.

## 12.2 Progettazione dell'interfaccia

Dovrebbe essere chiaro ora, che scrivere un sistema operativo moderno non è facile; ma da dove si deve iniziare? Probabilmente il miglior modo per iniziare è pensare alle interfacce che esso fornisce: un sistema operativo fornisce un insieme di servizi, la maggior parte sotto forma di tipi di dato (esempio file) e le operazioni su di essi (esempio read); il loro insieme forma l'interfaccia verso gli utenti. Si noti che in questo contesto gli utenti del sistema operativo sono i programmati che scrivono codice che utilizza chiamate di sistema, non persone che utilizzano i programmi applicativi.

In aggiunta alle interfacce principali per le chiamate di sistema, la maggior parte dei sistemi operativi hanno interfacce addizionali. Per esempio, alcuni programmati hanno la necessità di scrivere driver di dispositivi da inserire nel sistema operativo: questi driver vedono certe caratteristiche e possono effettuare certe chiamate di procedura, pertanto anche tali caratteristiche e chiamate definiscono un'interfaccia, ma molto differente da quella che vedono i programmati delle applicazioni. Tutte queste interfacce devono essere progettate accuratamente se il sistema possa avere successo.

### 12.2.1 Principi guida

Ci sono principi che possono guidare i progettisti di un'interfaccia? Crediamo di sì: riassunti brevemente essi sono la semplicità, la completezza e la possibilità di essere implementati in maniera efficiente.

#### Principio 1: Semplicità

Una interfaccia semplice è più facile da comprendere e da implementare senza bachi. Tutti i progettisti di sistemi dovrebbero studiare a memoria questa famosa citazione del francese Antoine de St. Exupéry, pioniere dell'aviazione e scrittore:

*La perfezione è raggiunta non quando non c'è più nulla da aggiungere, ma quando non c'è più nulla da togliere.*

Questo principio afferma che poco è meglio di troppo, almeno nei sistemi operativi. Un altro modo per dire questo è il principio di KISS: Keep It Simple, Stupid (fallo semplice, stupido).

#### Principio 2: Completezza

Naturalmente, l'interfaccia deve rendere possibile fare ogni cosa di cui gli utenti hanno la necessità, cioè, deve essere completa. Questo ci porta ad un'altra famosa citazione, quella di Albert Einstein:

*Ogni cosa dovrebbe essere più semplice possibile, ma non più semplice.*

In altre parole, il sistema operativo dovrebbe fare esattamente ciò che è necessario e non di più. Se gli utenti hanno bisogno di memorizzare dati, esso deve fornire qualche meccanismo per la memorizzazione dei dati; se gli utenti hanno bisogno di comunicare tra di loro, il sistema operativo deve fornire un meccanismo di comunicazione, e così via. Nel-

la sua conferenza per il premio Turing del 1991, Fernando Corbató, uno dei progettisti di CTSS e di MULTICS, combiò il concetto di semplicità e di completezza e disse:

*Prima di tutto, è importante enfatizzare il valore di semplicità e di eleganza, in quanto la complessità è un modo per creare difficoltà e, come abbiamo visto, per generare errori. La mia definizione di eleganza è il raggiungimento di una data funzionalità con il minimo meccanismo e la massima chiarezza.*

L'idea chiave qui è il *minimo meccanismo*. In altre parole, ogni caratteristica, funzione e chiamata di sistema dovrebbe avere il proprio peso, dovrebbe fare una cosa e farla bene. Quando un membro di un gruppo di progetto propone di estendere una chiamata di sistema o di aggiungere qualche nuova caratteristica, gli altri dovrebbero porre la domanda: "Succederebbe qualcosa di terribile se la togliessimo?". Se la risposta è: "No, ma qualcuno, un giorno, potrebbe trovare questa caratteristica molto utile", la si mette in una libreria a livello utente, non nel sistema operativo, anche se quella strada è più lenta. Non tutte le caratteristiche devono essere più veloci di un proiettile: l'obiettivo è quello di mantenere quello che Corbató chiama minimo meccanismo.

Consideriamo brevemente due esempi dalla mia esperienza personale: MINIX e Amoeba. Per tutti gli intenti e scopi, MINIX ha tre chiamate di sistema: `send`, `receive` e `sendrec`. Il sistema è strutturato come una collezione di processi, in cui il gestore della memoria, il file system ed ogni driver di dispositivo è un processo schedulabile separato. Da una prima approssimazione, tutto quello che fa il nucleo è di schedulare i processi e gestire lo scambio dei messaggi tra loro. Di conseguenza, sono necessarie solo due chiamate di sistema: `send`, per spedire un messaggio e `receive` per riceverne uno. La terza chiamata `sendrec` è semplicemente un'ottimizzazione per ragioni di efficienza che consente di spedire un messaggio e di richiedere la risposta, con una sola trap di kernel. Ogni altro servizio viene effettuato richiedendo a qualche altro processo (per esempio, il processo file system o il driver del disco) di eseguire il lavoro.

Amoeba è ancora più semplice. Ha una sola chiamata di sistema: eseguire la chiamata di procedura remota. Questa chiamata spedisce un messaggio ed attende una risposta; è essenzialmente la `sendrec` di MINIX, ogni altro servizio è costruito su questa chiamata.

### Principio 3: Efficienza

La terza linea guida è l'efficienza dell'implementazione: se una caratteristica o una chiamata di sistema non può essere implementata efficientemente, probabilmente non vale la pena di averla. Dovrebbe anche essere intuitivamente ovvio al programmatore quanto costa una chiamata di sistema; per esempio, i programmati di UNIX presumono che la chiamata di sistema `lseek` sia meno costosa della chiamata di sistema `read` in quanto la prima cambia solamente un puntatore in memoria, mentre la seconda esegue un I/O su disco. Se l'intuizione del costo è sbagliata, i programmati scriveranno programmi inefficienti.

## 12.2.2 Paradigmi

Una volta che gli obiettivi sono stati stabiliti, il progetto può iniziare, ed un buon punto di partenza è quello di pensare a come i clienti vedranno il sistema. Una delle questioni più importanti è come rendere tutte le caratteristiche del sistema ben unite tra loro e presentare quella che spesso è chiamata **coerenza architettonica**. A questo proposito, è importante distinguere due tipi di "clienti" dei sistemi operativi. Da una parte ci sono gli

*utenti*, che interagiscono con i programmi applicativi; dall'altra parte ci sono i *programmatori*, che li scrivono. I primi generalmente hanno a che fare con la GUI; i secondi generalmente hanno a che fare con l'interfaccia per le chiamate di sistema. Se l'intenzione è di avere una singola GUI che riguarda l'intero sistema, come ad esempio nel Macintosh, il progetto dovrebbe iniziare lì; se, invece, l'intenzione è di supportare diverse possibili GUI, come in UNIX, dovrebbe essere progettata per prima l'interfaccia per le chiamate di sistema. Realizzare per prima la GUI significa essenzialmente fare una progettazione top-down ed i problemi sono: quali caratteristiche avrà, come l'utente interagirà con essa, e come dovrebbe essere progettato il sistema per supportarla? Per esempio, se la maggior parte dei programmi visualizzano le icone nello schermo e quindi attendono che l'utente clicchi su una di loro, questo suggerisce un modello basato sugli **eventi** (event-driven) per la GUI e probabilmente anche per il sistema operativo. D'altra parte, se lo schermo è quasi sempre pieno di finestre di testo, allora probabilmente sarà migliore un modello in cui i processi leggono dalla tastiera.

Realizzare per prima l'interfaccia per le chiamate di sistema costituisce una progettazione bottom-up i cui problemi riguardano quali tipi di caratteristiche sono generalmente necessarie per i programmatori. In effetti, non sono necessarie molte caratteristiche speciali per supportare una GUI: per esempio, X, il sistema a finestre di UNIX, è solo un grosso programma C che effettua `read` e `write` sulla tastiera, sul mouse e sullo schermo. X è stato sviluppato molto tempo dopo UNIX, e non ha richiesto molti cambiamenti nel sistema operativo per svolgere il suo compito; questa esperienza ha confermato il fatto che UNIX fosse sufficientemente completo.

## Paradigmi di interfaccia utente

Sia per l'interfaccia a livello GUI che per l'interfaccia per chiamate di sistema, l'aspetto più importante è quello di avere un buon paradigma (a volte chiamato metafora) per fornire un modo di vedere l'interfaccia. Molte GUI per computer da tavolo utilizzano il paradigma WIMP che abbiamo discusso nel Capitolo 5; questo paradigma utilizza puntare-e-fare click, puntare-e-fare doppio-click, trascinare, e altri idiomati in tutta l'interfaccia per fornire nel complesso una coerenza architettonica. Spesso vi sono esigenze aggiuntive per i programmi, come ad esempio avere un menu a barra con FILE, VISUALIZZA, ed altre diciture, ciascuna delle quali ha un certo menu di voci ben conosciuto; in questo modo, gli utenti che conoscono un programma possono facilmente comprenderne un altro.

Tuttavia, l'interfaccia utente WIMP non è l'unica possibile: alcuni computer palmari utilizzano come interfaccia la scrittura manuale stilizzata; dispositivi multimediali dedicati possono utilizzare un'interfaccia stile videoregistratore, e certamente, l'input vocale ha un paradigma completamente differente. Quello che è importante non è tanto il paradigma scelto, ma il fatto che ci sia un unico paradigma di riscrittura che unifichi l'intera interfaccia utente.

Qualsiasi paradigma venga scelto, è importante che tutti i programmi applicativi lo utilizzino; di conseguenza, i progettisti del sistema devono fornire librerie e strumenti agli sviluppatori delle applicazioni, che diano loro l'accesso alle procedure che producono lo stile uniforme. Il progetto dell'interfaccia utente è molto importante, ma non è il soggetto di questo libro, quindi ora torneremo indietro al concetto di interfaccia dei sistemi operativi.

## Paradigmi di esecuzione

La coerenza architettonica è importante al livello utente, ma è ugualmente importante a livello di interfaccia per le chiamate di sistema; e a questo proposito, frequentemen-

```

(a) (b)

main()
{
 int ...;
 init(...);
 fai_qualcosa(...);
 read(...);
 fai_qualcosaltro(...);
 write(...);
 vai_sempre(...);
 exit(0);
}

main()
{
 mess_t msg;
 init();
 while(prendi_messaggio(&msg)) {
 switch (msg.tipo) {
 case 1: ...;
 case 2: ...;
 case 3: ...;
 }
 }
}

```

Figura 12.1 (a) Codice algoritmico. (b) Codice event-driven.

te è utile distinguere tra il paradigma di esecuzione e il paradigma dei dati, che noi tratteremo entrambi, partendo dal primo.

Sono molto diffusi due paradigmi di esecuzione: algoritmico ed event-driven (basato sugli eventi). Il **paradigma algoritmico** è basato sull'idea che un programma è avviato per eseguire qualche funzione che esso conosce in anticipo oppure che ottiene dai suoi parametri; ad esempio, tale funzione potrebbe essere compilare un programma, eseguire la distinta delle borse paga, o far volare un aereo a San Francisco. La logica di base è cablata nel codice del programma, che realizza, di tanto in tanto, chiamate di sistema per prendere l'input dell'utente, ottenere servizi dal sistema operativo, ecc. Questo approccio è delineato in Figura 12.1(a).

L'altro paradigma di esecuzione è il **paradigma event-driven** di Figura 12(b), in cui il programma esegue alcuni tipi di inizializzazione, per esempio visualizzando una certa schermata, e quindi attendendo che il sistema operativo gli dica qualcosa sul primo evento, che è spesso la pressione di un tasto o un movimento del mouse. Questo metodo è utile per programmi altamente interattivi.

Ognuno di questi modi di realizzare il lavoro produce un proprio stile di programmazione. Nel paradigma algoritmico, gli algoritmi sono centrali e il sistema operativo è visto come un fornitore di servizi; nel paradigma event-driven, il sistema operativo fornisce anche servizi, ma questo ruolo è offuscato dal suo ruolo di coordinatore delle attività dell'utente e di generatore di eventi che sono consumati dai processi.

## Paradigma dei dati

Il paradigma di esecuzione non è l'unico paradigma tirato fuori dal sistema operativo; l'altro, ugualmente importante, è il **paradigma dei dati**. La questione chiave qui è come sono presentate al programmatore le strutture e i dispositivi di sistema. Nei primi sistemi batch FORTRAN, ogni cosa era modellata come un nastro magnetico sequenziale: le schede da leggere erano considerate nastri di input, le schede pronte da perforare erano considerate nastri di output, e l'output per la stampante era considerato un nastro di output; anche i file su disco erano considerati come nastri, e l'accesso

casuale ad un file era possibile solo riavvolgendo il nastro corrispondente al file e leggendolo nuovamente.

La mappatura era effettuata utilizzando schede di controllo dei job come queste:

```
MOUNT(NASTRO08, BOBINA781)
RUN(INPUT, MIEIDATI,OUTPUT,PERFORA, NASTRO08)
```

La prima scheda indica all'operatore di andare a prelevare il nastro della bobina 781 dal contenitore dei nastri e di montarlo nel dispositivo per nastro numero 8. La seconda scheda indica al sistema operativo di eseguire il programma FORTRAN appena compilato, mappando *INPUT* (cioè il lettore di schede) con il nastro logico 1, il file su disco *MIEIDATI* con il nastro logico 2, la stampante (chiamata *OUTPUT*) con il nastro logico 3, il perforatore di schede (chiamata *PERFORA*) con il nastro logico 4 e il dispositivo fisico per nastro numero 8 con il nastro logico 5.

FORTRAN aveva una sintassi per leggere e scrivere nastri logici: leggendo dal nastro logico 1, il programma prendeva una scheda in input; scrivendo sul nastro logico 3, l'output doveva successivamente apparire sulla stampante; leggendo dal nastro logico 5, la bobina di nastro 781 poteva essere letta, e così via. Si noti che l'idea del nastro era solo un paradigma per integrare il lettore di schede, la stampante, il perforatore, i file su disco e i nastri. In questo esempio, solo il nastro logico 5 era un nastro fisico, gli altri erano file su disco ordinari (nello spooler): era un paradigma primitivo, ma era un inizio nella giusta direzione.

Successivamente arrivò UNIX, che andò oltre utilizzando il modello di "ogni cosa è un file". Utilizzando questo paradigma, tutti i dispositivi di I/O sono considerati come file e possono essere aperti e manipolati come file ordinari. Le istruzioni C

```
fd1 = open ("file1", O_RDWR);
fd2 = open ("/dev/tty", O_RDWR);
```

aprono un file di disco vero e proprio, e il terminale utente. Le istruzioni successive possono utilizzare *fd1* e *fd2* per leggerli e scriverli, rispettivamente; da questo punto in poi, non c'è differenza tra accedere al file ed accedere al terminale, ad eccezione del fatto che i posizionamenti (seek) sul terminale non sono consentiti.

UNIX non solo unificò i file e i dispositivi di I/O, ma consentì anche di accedere agli altri processi all'interno delle pipe come se fossero file; inoltre, quando sono supportati i file mappati, un processo può considerare la sua memoria virtuale come fosse un file. Infine, nelle versioni di UNIX che supporta il file system */proc*, l'istruzione C

```
fd3 = open("/proc/501", O_RDWR);
```

consente al processo di (cercare di) accedere alla memoria del processo 501 per letture e scritture, utilizzando il descrittore di file *fd3*, cosa a volte utile, ad esempio per un debugger.

Windows 2000 va ancora oltre e cerca di considerare ogni cosa come un oggetto: una volta che un processo ha acquisito una descrittore (handle) valido per un file, un processo, un semaforo, una casella di posta, o per altri oggetti del nucleo, può effettuare operazioni su di esso. Questo paradigma è ancora più generale di quello di UNIX e molto più generale di quello di FORTRAN.

L'unificazione dei paradigmi avviene anche in altri contesti: vale la pena citarne uno: il Web. Il paradigma sottostante al Web afferma che il ciberspazio è pieno di documenti ognuno dei quali ha un URL; digitando un URL o cliccando su un riferimento al suo URL,

si ottiene il documento. In realtà, molti "documenti" non sono per nulla dei documenti, ma sono generati da un programma o da una script di shell quando arriva una richiesta. Per esempio, quando un utente chiede ad un venditore on-line una lista di CD di un particolare artista, il documento è generato al volo da un programma; certamente non esiste prima che la richiesta fosse effettuata.

Abbiamo quindi visto quattro casi: cioè, ogni cosa è un nastro, file, oggetto o documento, ed in tutti e quattro i casi, l'intenzione è di unificare dati, dispositivi e altre risorse per renderli semplici da gestire. Ogni sistema operativo dovrebbe avere un tale paradigma unificante dei dati.

### 12.2.3 L'interfaccia per le chiamate di sistema

Se uno crede nell'affermazione di Corbató del minimo meccanismo, allora il sistema operativo dovrebbe fornire tante chiamate di sistema quante bastano per poter girare, ed ognuna dovrebbe essere più semplice possibile (ma non più semplice di così). Un paradigma dei dati unificante può giocare un ruolo fondamentale di aiuto in questo contesto: per esempio, se file, processi, dispositivi di I/O e molte altre cose, vengono tutti considerati come file o oggetti, allora possono essere letti con una singola chiamata di sistema `read`, altrimenti, potrebbe essere necessario avere, tra le altre, chiamate separate per `read_file`, `read_proc` e `read_tty`.

In alcuni casi, può sembrare che le chiamate di sistema abbiano bisogno di diverse varianti, ma spesso è più pratico avere una chiamata di sistema che gestisce il caso generale, e diverse procedure di libreria per nascondere questo fatto ai programmati. Per esempio, UNIX ha una chiamata di sistema per rimpiazzare uno spazio di indirizzamento virtuale di un processo, `exec`. La chiamata più generale è

```
exec(nome, argp, envp)
```

che carica il file eseguibile *nome* e gli fornisce gli argomenti a cui punta tramite *argp* e le variabili d'ambiente a cui punta tramite *envp*. A volte è conveniente elencare gli argomenti esplicitamente, in questo modo la libreria contiene le procedure che vengono chiamate come segue:

```
exec1(nome, arg0, arg1, ..., argn, 0);
execle(nome, arg0, arg1, ..., argn, envp);
```

Tutto quello che fanno queste procedure è di inserire gli argomenti in un array e quindi chiamare `exec` per eseguire il lavoro. Questo metodo è il migliore al mondo: una semplice singola chiamata di sistema mantiene il sistema operativo semplice, ed in più il programmatore ha la convenienza di avere diversi modi di chiamare `exec`.

Di sicuro, cercare di avere una chiamata per gestire ogni caso possibile può facilmente scappare di mano; in UNIX, la creazione di un processo richiede due chiamate: `fork` seguita da `exec`, dove la prima non ha parametri e la seconda ha tre parametri. Per confronto, la chiamata API di Win 32 per la creazione di un processo, `CreateProcess`, ha 10 parametri, uno dei quali è un puntatore ad una struttura con 18 parametri aggiuntivi.

Molto tempo fa, qualcuno avrebbe dovuto fare la domanda: "Accadrebbe qualcosa di terribile se noi togliessimo alcuni di questi parametri?". La risposta sincera sarebbe stata: "In alcuni casi i programmati potrebbero avere da fare più lavoro per raggiungere un particolare obiettivo, ma il risultato netto sarebbe un sistema operativo più semplice, più piccolo e più affidabile." Certamente, le persone che hanno proposto la versione 10+18

parametri, potrebbero avere aggiunto: "Ma agli utenti piacciono tutte queste caratteristiche" e la risposta a questo potrebbe essere stata: "A loro piacciono ancora di più sistemi che utilizzano poca memoria e non cadono mai". I compromessi tra avere più funzionalità al costo di più memoria almeno sono visibili e hanno un prezzo (poiché il prezzo di memoria è conosciuto); tuttavia, è difficile stimare le cadute di sistema supplementari annuali, dovute alle caratteristiche aggiunte, e stimare se gli utenti farebbero le stesse scelte se conoscessero il prezzo nascosto. Questo effetto può essere riassunto nella prima legge di Tanenbaum del software:

*Aggiungere più codice aggiunge più bachi.*

Aggiungere più caratteristiche aggiunge più codice e perciò aggiunge più bachi. I programmatore che credono che aggiungere nuove caratteristiche non aggiunge nuovi bachi sono novellini del computer oppure credono che le fate li proteggeranno.

La semplicità non è il solo problema che si presenta quando si progettano le chiamate di sistema. Una importante considerazione è lo slogan di Lampson (1984):

*Non nascondere la potenza.*

Se l'hardware ha un modo estremamente efficiente di eseguire qualcosa, dovrebbe essere reso visibile al programmatore in modo semplice e non sepolto dentro qualche altra astrazione: l'obiettivo delle astrazioni è di nascondere le proprietà indesiderabili, non di nascondere quelle desiderabili. Per esempio, supponiamo che l'hardware abbia un modo speciale di muovere grosse mappe di bit per lo schermo (ad esempio la RAM del video) ad alta velocità. Sarebbe plausibile avere una nuova chiamata di sistema per accedere a questo meccanismo, invece di fornire solo modi di leggere la RAM del video nella memoria principale e di riscriverla all'indietro; la nuova chiamata dovrebbe solo muovere bit e nient'altro. Se una chiamata di sistema è veloce, gli utenti possono sempre costruire interfacce più convenienti su di essa; se è lenta, nessuno la userà.

Un altro problema di progettazione riguarda le chiamate orientate alla connessione piuttosto che le chiamate prive di connessione. Le chiamate di sistema standard per UNIX e per Win32 per leggere un file sono orientate alla connessione: prima si apre il file, quindi si legge, infine si chiude, ed anche alcuni protocolli di accesso a file remoti sono orientati alla connessione. Per esempio, per usare FTP, l'utente per prima cosa si collega alla macchina remota, legge i file, e quindi si scollega.

D'altra parte, alcuni protocolli di accesso ai file remoti non sono orientati alla connessione: NFS è privo di connessione, come abbiamo visto nel Capitolo 10, ed ogni chiamata di NFS è indipendente, quindi i file non vengono aperti prima di effettuare letture o scritture, ed inoltre, non devono essere chiusi successivamente. Il Web è anch'esso privo di connessione: per leggere una pagina Web si deve solo chiederla; non c'è nessuna richiesta precedente di installazione (è richiesta una connessione TCP, ma questa avviene ad un livello più basso del protocollo; il protocollo HTTP per accedere al Web è esso stesso privo di connessione).

Il compromesso tra ogni meccanismo orientato alla connessione ed uno privo di connessione è il lavoro aggiuntivo richiesto per fornire il meccanismo (ad esempio aprire i file), e il vantaggio di non doverlo fare su (possibilmente molte) chiamate successive. Per fare I/O su file in una macchina singola, dove il costo di inizializzazione è basso, probabilmente il metodo standard (prima apri, poi usa) è il metodo migliore; per file system remoti, possono essere difesi entrambi i metodi.

Un altro problema relativo alla interfaccia per le chiamate di sistema è la visibilità. La lista delle chiamate del sistema obbligatorie in POSIX è facile da trovare, tutti i sistemi

UNIX la supportano, oltre ad un piccolo numero di chiamate aggiuntive, e tale lista completa è sempre pubblica. Al contrario, Microsoft non ha mai reso pubbliche le chiamate di sistema di Windows 2000; invece, sono state rese pubbliche le API Win 32, ed altre API, che contengono un vasto numero di chiamate di libreria (più di 13000 per Windows 2000), di cui solo un piccolo numero sono vere chiamate di sistema. Il vantaggio di rendere pubbliche tutte le chiamate di sistema è di consentire ai programmati di conoscere cosa è conveniente (funzioni eseguite nello spazio utente) e cosa è dispendioso (chiamate del nucleo); il vantaggio di non renderle pubbliche è che dà agli implementatori la flessibilità di cambiare le vere e proprie chiamate di sistema sottostanti per renderle migliori, senza interrompere i programmi utente.

## 12.3 Implementazione

Abbandoniamo l'utente, e le chiamate di sistema, e diamo ora un'occhiata a come implementare un sistema operativo. Nelle otto sezioni successive esamineremo alcuni problemi generali di concetto relativi alle strategie di implementazione, quindi daremo un'occhiata ad alcune tecniche di basso livello che sono spesso utili.

### 12.3.1 Struttura del sistema

Probabilmente la prima decisione che devono prendere gli implementatori riguarda come dovrebbe essere la struttura del sistema; nella Sezione 1.7 abbiamo esaminato le principali possibilità, ma qui le rivedremo, in quanto un progetto monolitico non strutturato non è, realmente, una buona idea, fatta eccezione, forse, per un piccolo sistema operativo all'interno, ad esempio, di un frigorifero, ma persino in questo caso è discutibile.

#### Sistemi a strati

Un approccio ragionevole che si è affermato negli anni è il sistema a strati. Il sistema THE di Dijkstra (Figura 1.25) fu il primo sistema operativo a strati; anche UNIX (Figura 10.3) e Windows 2000 (Figura 11.7) hanno una struttura a strati, ma in entrambi la stratificazione è un modo per cercare di descrivere il sistema, più che un vero principio guida utilizzato nella realizzazione del sistema stesso.

Per un sistema nuovo, i progettisti che decidono di seguire questa strada, *prima di tutto*, dovrebbero scegliere molto attentamente gli strati, e definire la funzionalità di ciascuno di essi. Lo strato più basso dovrebbe sempre cercare di nascondere le peggiori peculiarità dell'hardware, come fa HAL in Figura 11.7; lo strato successivo, probabilmente, dovrebbe gestire le interruzioni, i cambi di contesto e la MMU, in modo che, sopra questo strato, il codice sia generalmente indipendente dalla macchina. Da questo livello in poi, i diversi progettisti avranno diverse preferenze (e pregiudizi); una possibilità è quella di avere lo strato 3 che gestisce i thread, compresa la schedulazione e la sincronizzazione all'interno di un thread, come mostrato in Figura 12.2. In tale caso, l'idea è che, partendo dal livello 4, si hanno dei thread opportuni, che sono schedulati normalmente, e che vengono sincronizzati utilizzando un meccanismo standard (per esempio i mutex).

Nello strato 4 possiamo trovare i driver dei dispositivi, ognuno considerato come un thread separato, con il proprio stato, il proprio contatore di programma, i propri registri, eccetera, possibilmente (ma non necessariamente) all'interno dello spazio di indirizzamento del nucleo. Tale progetto può semplificare notevolmente la struttura di I/O, poi-

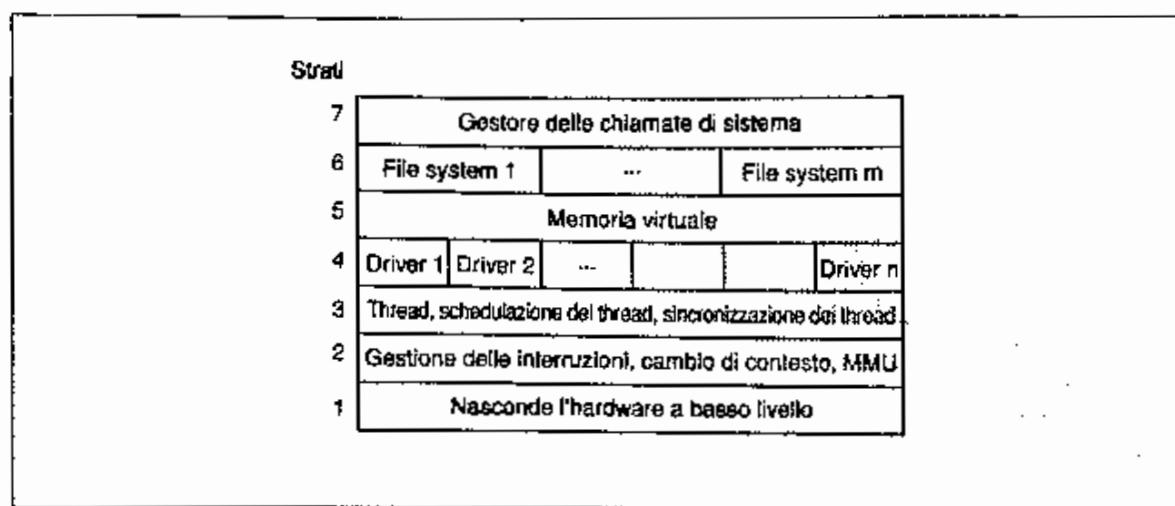


Figura 12.2 Un possibile progetto per un moderno sistema operativo a strati.

ché quando avviene un'interruzione, la si può convertire in una unlock su un mutex, ed una chiamata allo scheduler per schedulare (potenzialmente) i nuovi processi ready, che erano bloccati dal mutex. MINIX utilizza questo approccio, ma in UNIX, Linux e Windows 2000, i gestori delle interruzioni girano in una specie di limbo, piuttosto che come veri e propri thread, che possono essere schedulati, sospesi, eccetera. Poiché gran parte della complessità di ogni sistema operativo è nell'I/O, ogni tecnica per renderlo più gestibile ed encapsulato, è degna di considerazione.

Sopra il livello 4, potremmo aspettarci di trovare la memoria virtuale, uno o più file system, e i gestori delle chiamate di sistema. Se la memoria virtuale è ad un livello più basso del file system, allora l'intera cache dei blocchi può essere paginata, consentendo al gestore della memoria virtuale di determinare dinamicamente come la memoria reale, compresa la cache, dovrebbe essere divisa tra pagine utente e pagine nucleo; Windows 2000 segue questa strada.

## Exokernel

Mentre la gestione a strati ha i suoi sostenitori tra i progettisti dei sistemi operativi, c'è un'altra fazione che ha una visione completamente opposta (Engler ed altri, 1995). La loro visione è basata sul principio **end-to-end** (Saltzer ed altri, 1984); questo concetto afferma che se qualcosa deve essere fatto proprio dal programma utente, allora è inutile farlo anche in uno strato più basso.

Consideriamo una applicazione di questo principio all'accesso ad un file remoto: se un sistema è preoccupato del fatto che i dati possano essere alterati durante il loro percorso, esso dovrebbe fare in modo che su ogni file venga effettuato il controllo di parità al momento in cui viene scritto, memorizzandone il risultato insieme con il file. Quando un file viene trasferito via rete dal disco sorgente al processo di destinazione, il controllo di parità viene anch'esso trasferito, e rieseguito anche sul ricevente; quindi, se i due non coincidono, il file viene scartato e trasferito nuovamente.

Questo controllo è più accurato dell'utilizzo di un protocollo di rete affidabile, poiché esso cattura anche gli errori del disco, gli errori della memoria, gli errori del software nel router, e altri errori, oltre a quelli di trasmissione dei dati. Il principio end-to-end afferma, quindi, che non è necessario utilizzare un protocollo di rete affidabile, poiché il destinatario finale (il processo ricevente) ha abbastanza informazioni per verificare la corret-

tezza del file stesso; l'unica ragione per utilizzare un protocollo di rete affidabile, in questo contesto, è l'efficienza, cioè consentire la cattura e la correzione degli errori di trasmissione il più presto possibile.

Il principio end-to-end può essere esteso a quasi tutti i componenti dei sistemi operativi; il suo concetto chiave è che il sistema operativo non deve effettuare nulla che il programma utente possa effettuare da solo. Per esempio, perché avere un file system? Si può lasciare all'utente soltanto la possibilità di leggere e scrivere una porzione del disco fisico in maniera protetta; di sicuro, alla maggior parte degli utenti piace avere i file, ma il principio end-to-end afferma che il file system dovrebbe essere una libreria di procedure collegate a ciascun programma che abbia bisogno di utilizzare i file; quindi, questo approccio consente a programmi differenti di avere file system differenti. Questa linea di ragionamento afferma che tutto ciò che il sistema operativo dovrebbe fare è allocare le risorse (per esempio la CPU e i dischi) tra gli utenti in competizione, garantendo la sicurezza. Exokernel è un sistema operativo costruito in accordo con il principio end-to-end (Engler ed altri, 1995).

## Sistemi client-server

Un compromesso tra avere un sistema operativo che fa qualsiasi cosa e avere un sistema operativo che non fa nulla, è quello di avere un sistema operativo che fa qualche cosa; questo modello porta ad avere un microkernel, mentre la maggior parte del sistema operativo gira come processo server a livello utente, come illustrato in Figura 1.27. Questo è il più modulare e flessibile di tutti i progetti, e la scelta estrema, per quanto riguarda la flessibilità, è quella di avere anche tutti i driver dei dispositivi che girano come processi utente, completamente protetti dal nucleo e dagli altri driver. Portare i driver fuori dal nucleo, eliminerebbe la più grande causa di instabilità in ogni sistema operativo — driver forniti da terzi contenenti errori — e sarebbe una enorme vittoria in termini di affidabilità.

Di sicuro, i driver dei dispositivi hanno la necessità di accedere ai registri dei dispositivi hardware, quindi è necessario qualche meccanismo che fornisca questa possibilità: se l'hardware lo consente, ogni processo driver potrebbe avere accesso solo a quei dispositivi di I/O di cui ha bisogno. Per esempio, con l'I/O a memoria mappata, ogni processo driver potrebbe avere mappata la pagina per il suo dispositivo, ma nessun'altra pagina di altri dispositivi; quindi, se lo spazio delle porte di I/O può essere parzialmente protetto, si potrebbe rendere disponibile a ciascun driver la sua porzione corretta di spazio delle porte.

Se anche non fosse disponibile nessuna facilitazione hardware, l'idea può ancora essere messa in pratica; quello che è necessario, quindi, è una nuova chiamata di sistema, disponibile solo ai processi dei driver dei dispositivi, che fornisca una lista di coppie del tipo (porta, valore). Quello che fa il nucleo, innanzitutto, è controllare se il processo possiede tutte le porte nella lista; se è così, copia i valori corrispondenti sulle porte, per inizializzare i dispositivi di I/O. Una chiamata analoga può essere utilizzata per leggere le porte di I/O in maniera protetta.

Questo approccio impedisce ai driver dei dispositivi di esaminare (e danneggiare) le strutture dati del nucleo, e questa è (in gran parte) una buona cosa. Un insieme di chiamate analogo potrebbe essere reso disponibile per consentire ai processi driver di leggere e scrivere le tabelle del nucleo, ma soltanto in maniera controllata e con l'approvazione del nucleo.

Il problema principale di questo approccio, e dei microkernel in generale, è il crollo delle prestazioni causato da tutti i cambi di contesto aggiuntivi, tuttavia, praticamente tutto lo studio dei microkernel risale a molti anni fa, quando le CPU erano molto più len-

te; oggi, le applicazioni che utilizzano ogni goccia di potenza della CPU e che non possono tollerare una piccola perdita di prestazioni, sono poche e capitano assai di rado. Dopo tutto, quando gira un programma di videoscrittura o un browser Web, la CPU è probabilmente inattiva al 90% del tempo, quindi, se un sistema operativo basato sul microkernel trasforma un sistema inaffidabile a 900 MHz, in un sistema affidabile a 800 MHz, probabilmente pochi utenti si lamenterebbero; dopo tutto, molti di loro erano proprio contenti, solo pochi anni fa, dei loro precedenti computer, che marciavano alla velocità allora stupenda di 100 MHz.

## Sistemi estensibili

Con i sistemi client-server discussi precedentemente, l'idea era quella di portare fuori dal nucleo tutto quello che era possibile; l'approccio opposto è quello di inserire più moduli nel nucleo, ma in maniera protetta, e di sicuro la parola chiave, qui, è *protetto*. Abbiamo studiato alcuni meccanismi di protezione nella Sezione 9.5.6, che erano pensati, inizialmente, per l'importazione di applet da Internet, ma che sono applicabili nello stesso modo per l'inserimento di codice esterno nel nucleo; i più importanti sono il *sandboxing* e la marcatura del codice, in quanto l'interpretazione non è molto pratica per il codice del nucleo.

Naturalmente, un sistema estensibile, da solo, non è un buon modo di strutturare un sistema operativo; tuttavia, si può costruire un nuovo sistema operativo per l'applicazione che ci interessa, cominciando da un sistema minimale, fatto da poco più che un meccanismo di protezione, e aggiungendo al nucleo dei moduli protetti, uno alla volta, fino a raggiungere la funzionalità desiderata. In questo senso, un nuovo sistema operativo può essere adattato ad ogni applicazione, comprendendo solo le parti che richiede. *Paramecium* è un esempio di tale sistema (Van Doorn, 2001).

## Thread del nucleo

Un altro problema rilevante, è quello dei thread di sistema, indipendentemente da quale modello di strutturazione si sia scelto. Talvolta conviene consentire che esistano thread nel nucleo, separati dai processi utente; questi thread possono girare in background, per scrivere pagine sporche su disco, scambiare processi tra la memoria principale e il disco, e così via. In pratica il nucleo stesso può essere interamente composto da tali thread, in modo che, se un utente effettua una chiamata di sistema, invece di eseguire il thread dell'utente in modalità kernel, si blocca il thread dell'utente, e si passa il controllo al thread del nucleo, che subentra ed esegue il lavoro.

In aggiunta ai thread del nucleo che girano in background, la maggior parte dei sistemi operativi fa partire anche molti processi demoni in background. Sebbene questi non facciano parte del sistema operativo, spesso eseguono attività tipiche "del sistema", che possono riguardare la ricezione e la spedizione di e-mail, ed il servizio di varie richieste da parte di utenti remoti, come ad esempio FTP e pagine Web.

### 12.3.2 Meccanismi contro politiche

Un altro principio che aiuta la coerenza architetturale, oltre a mantenere i sistemi piccoli e ben strutturati, è quello della separazione tra meccanismi e politiche. Inserendo i meccanismi nel sistema operativo, e lasciando le politiche ai processi utente, il sistema stesso può rimanere invariato, anche se vi è la necessità di cambiare le politiche. Anche

volendo lasciare nel nucleo i moduli delle politiche, essi dovrebbero essere isolati dai meccanismi, se possibile, in modo che i cambiamenti nei moduli delle politiche non intacchino i moduli dei meccanismi.

Per rendere maggiormente chiara la divisione tra politiche e meccanismi, consideriamo due esempi del mondo reale: come primo esempio, consideriamo una grande azienda che ha un ufficio libro paga, con l'incarico di pagare gli stipendi ai dipendenti. L'ufficio possiede dei computer, del software, dei libretti di assegni in bianco, conclude accordi con le banche, e possiede altri meccanismi per realizzare effettivamente il pagamento degli stipendi. Tuttavia, la politica - che determina chi deve essere pagato e quanto - è completamente separata ed è decisa dalla direzione, quindi, l'ufficio libro paga deve soltanto eseguire cosa gli viene detto di fare.

Come secondo esempio, consideriamo un ristorante; questo possiede i meccanismi per servire i clienti, come i tavoli, i vassoi, i camerieri, una cucina attrezzata; conclude accordi con le aziende che emettono carte di credito, e così via. La politica è stabilita dallo chef, e precisamente riguarda cosa inserire nel menù: se lo chef decide che il tofu non è più di moda mentre le bistecche di carne lo sono, questa nuova politica può essere gestita dal meccanismo esistente.

Ora consideriamo alcuni esempi tratti dai sistemi operativi: prima di tutto, consideriamo la schedulazione dei thread. Il nucleo potrebbe avere uno scheduler con priorità, con  $k$  livelli di priorità; il meccanismo è un array, indicizzato da un livello di priorità, come mostrato in Figura 10.11 o in Figura 11.19, dove ogni ingresso è la testa di una lista di thread ready a quel livello di priorità, e lo scheduler si limita a scorrere l'array dalla priorità più alta alla priorità più bassa, selezionando i primi thread che incontra. La politica stabilisce le priorità: il sistema, ad esempio, può avere diverse classi di utenti, ognuna con una differente priorità. Esso può anche consentire ai processi utente di stabilire la priorità relativa dei propri thread; le priorità potrebbero aumentare dopo avere completato l'I/O, o diminuire dopo aver completato il quanto di tempo. Si potrebbero seguire numerose altre politiche, ma l'idea, qui, è la separazione tra lo stabilire la politica e l'eseguirla.

Un secondo esempio è la paginazione: il meccanismo comprende la gestione della MMU, mantenendo liste di pagine occupate e liste di pagine libere, ed il codice per alternare le pagine da e per il disco. La politica si occupa di decidere cosa fare quando avviene un fault di pagina; la decisione potrebbe essere locale o globale, basata su un algoritmo LRU o basata su un algoritmo FIFO, o qualcos'altro, ma questo algoritmo può (e dovrebbe) essere completamente separato dai meccanismi di gestione effettiva delle pagine.

Un terzo esempio è quello di consentire ai moduli di essere caricati nel nucleo. Il meccanismo riguarda come vengono inseriti, come sono collegati, quali chiamate possono effettuare, e quali chiamate possono essere fatte loro; la politica consiste nel determinare a chi è consentito caricare un modulo nel nucleo, e quali moduli caricare. Forse solo il superuser può caricare i moduli, o forse ogni utente può caricare un modulo che è stato firmato digitalmente dall'autorità appropriata.

### 12.3.3 Ortogonalità

Un buon progetto di un sistema è composto da concetti separati, che possono essere combinati in maniera indipendente. Per esempio, in C, vi sono dei tipi di dato primitivi come gli interi, i caratteri e i numeri in virgola mobile (floating-point); inoltre, vi sono dei meccanismi per combinare i tipi di dato, come array, strutture ed unioni. Questi concetti vengono combinati in maniera indipendente, consentendo array di interi, array di caratteri, numeri floating-point come membri di strutture e di unioni, eccetera; infatti, una volta che è stato definito un nuovo tipo di dato, come ad esempio un array di interi, questo può

essere utilizzato come fosse un tipo di dato primitivo, per esempio come un membro di una struttura o di un'unione. L'abilità di combinare concetti separati in maniera indipendente, è chiamata **ortogonalità**, ed è una diretta conseguenza dei principi di semplicità e completezza.

Il concetto di ortogonalità è presente anche nei sistemi operativi, in varie forme. Un esempio è la chiamata di sistema di Linux `clone`, che crea un nuovo thread; questa chiamata ha come parametro una mappa di bit, che consente di condividere, oppure di copiare, lo spazio di indirizzamento, le directory di lavoro, i descrittori di file, ed i segnali, selezionandoli ad uno ad uno. Se si copia tutto, si ha un nuovo processo, come per la `fork`; se non si copia nulla, si crea un nuovo thread nel processo corrente. Tuttavia, è anche possibile creare forme intermedie di condivisione, non consentite nei sistemi UNIX tradizionali; quindi, separando le varie caratteristiche e rendendole ortogonali, è possibile un livello di controllo più accurato.

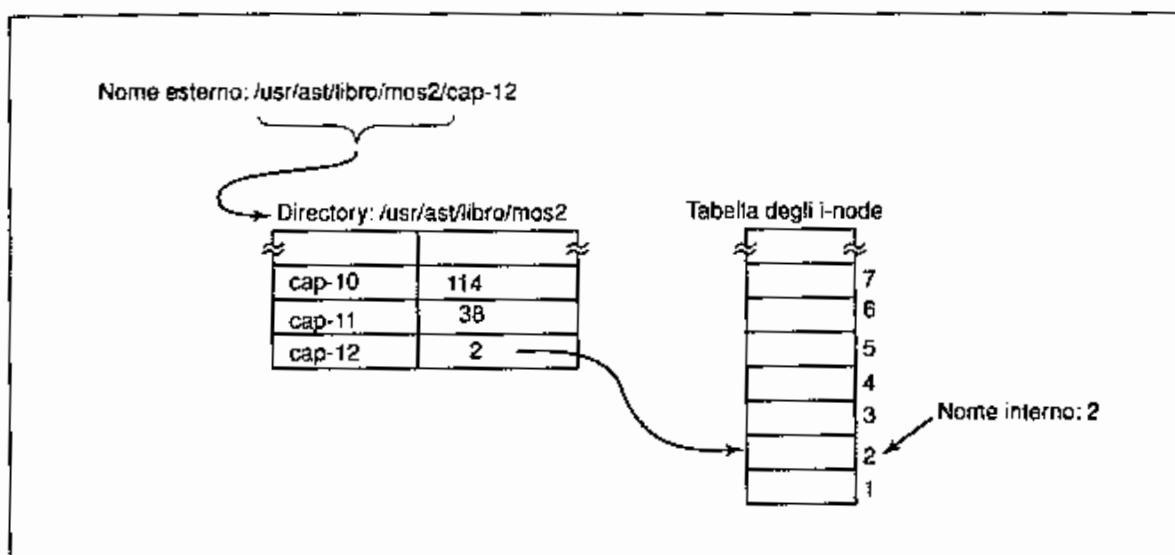
Un altro modo di usare l'ortogonalità è la separazione, in Windows 2000, del concetto di processo dal concetto di thread. Un processo è un contenitore di risorse, niente di meno e niente di più; un thread è un'entità schedulabile. Quando un processo riceve un handle per un altro processo, non importa quanti thread possiede; quando un thread è schedulato, non ha importanza a quale processo appartenga: i due concetti sono ortogonali.

Il nostro ultimo esempio di ortogonalità arriva da UNIX, dove la creazione dei processi viene effettuata in due passi: `fork` più `exec`. La creazione del nuovo spazio di indirizzamento, e il suo caricamento con una nuova immagine della memoria, vengono separati, consentendo di effettuare altre operazioni (come ad esempio la manipolazione dei descrittori dei file) fra la prima e la seconda chiamata di sistema. In Windows 2000, questi due passi non possono essere separati, cioè il concetto di creazione di un nuovo spazio di indirizzamento, e quello di caricamento di tale spazio non sono ortogonali. La sequenza `clone` più `exec`, in Linux, è ancora più ortogonale, poiché sono addirittura disponibili blocchi elementari di grana più fine. In generale, avere un numero piccolo di elementi ortogonali, che possano essere combinati in diversi modi, porta ad avere un sistema piccolo, semplice ed elegante.

### 12.3.4 Assegnazione dei nomi

La maggior parte delle strutture dati persistenti, utilizzate da un sistema operativo, ha un qualche tipo di nome o di identificatore per poter fare riferimento ad essa: esempi ovvi sono i nomi di login, i nomi dei file, i nomi dei dispositivi, gli ID dei processi, e così via. Il modo in cui questi nomi vengono costruiti e gestiti è una problematica importante nella progettazione e nell'implementazione di un sistema.

I nomi progettati per essere utilizzati dalle persone sono nomi formati da stringhe di caratteri in ASCII o in Unicode, e sono generalmente gerarchici. I percorsi (path) di una directory, ad esempio il percorso `/usr/ast/libro/mos2/cap-12`, sono chiaramente gerarchici, indicando una serie di directory da ricercare partendo dalla radice. Anche gli URL sono gerarchici: per esempio, `www.cs.vu.nl/~ast/` indica una macchina specifica (`www`) in un dipartimento specifico (`cs`) di una università specifica (`vu`) di un determinato paese (`nl`), e la parte successiva alla barra indica un file specifico sulla macchina designata; in questo caso, per convenzione, il file `www/index.html` nella home directory di `ast`. Si noti che gli URL (e gli indirizzi DNS in generale, compresi gli indirizzi di posta elettronica) sono scritti "all'indietro", partendo dal fondo dell'albero e salendo verso l'alto, a differenza dei nomi dei file, che partono dal vertice dell'albero e scendono verso il basso. Un altro modo per capire questi concetti è guardare se l'albero è scritto, partendo dal vertice, da sinistra verso destra, oppure da destra verso sinistra.



**Figura 12.3** Le directory vengono utilizzate per mappare i nomi esterni in nomi interni.

Spesso l'assegnazione dei nomi viene effettuata a due livelli: esterno ed interno. Per esempio i file, per essere utilizzati dalle persone, hanno sempre un nome, formato da una stringa di caratteri; inoltre, quasi sempre, essi hanno un nome interno utilizzato dal sistema. In UNIX, il nome vero e proprio di un file è il suo numero di i-node; il nome ASCII non è per niente utilizzato all'interno, anzi, il nome ASCII non è neppure unico, perché un file può avere diversi link ad esso. Analogamente il nome interno, in Windows 2000, è l'indice del file nella MFT: qui, il compito della directory è di fornire la mappatura tra il nome esterno ed il nome interno, come mostrato in Figura 12.3.

In molti casi (come nell'esempio del nome del file, visto sopra), il nome interno è un intero senza segno, che viene utilizzato come indice in una tabella del nucleo. Altri esempi di nomi che sono indici in tabelle sono i descrittori di file in UNIX e gli handle degli oggetti di Windows 2000: si noti che nessuno di questi ha una rappresentazione esterna, essi sono rigorosamente utilizzati dal sistema e dai processi in esecuzione. In generale, è una buona idea utilizzare indici di tabelle per i nomi transienti, che vengono persi quando il sistema viene rilanciato (reboot).

I sistemi operativi spesso supportano diversi spazi di nomi, sia esterni che interni. Per esempio, nel Capitolo 11, abbiamo visto tre spazi di nomi esterni supportati da Windows 2000: i nomi dei file, i nomi degli oggetti, e i nomi dei registri (c'è anche lo spazio dei nomi dell'Active Directory, che non avevamo considerato), oltre a questi, vi sono innumerevoli spazi di nomi interni che utilizzano interi senza segno, per esempio gli handle degli oggetti, le entrate della MFT, eccetera. Sebbene i nomi negli spazi dei nomi esterni siano tutte stringhe Unicode, cercare nel registro un nome di file non funziona, come pure utilizzare un indice di MFT nella tavola degli oggetti. In una buona progettazione, si riflette molto per valutare quanti spazi di nomi sono necessari, quale è la sintassi dei nomi in ciascuno di essi, come possono essere distinti, se esistono nomi relativi e assoluti, e così via.

### 12.3.5 Tempo di binding

Come abbiamo appena visto, i sistemi operativi utilizzano vari tipi di nomi per riferirsi agli oggetti; certe volte, la mappatura tra un nome ed un oggetto è fissa, ma qualche volta non lo è; in quest'ultimo caso può essere importante il momento in cui il nome è

legato all'oggetto. In generale, l'*early binding* (assegnazione in anticipo) è semplice, ma non è flessibile, mentre il *late binding* (assegnazione in ritardo) è più complicato ma spesso più flessibile.

Per chiarire il concetto di tempo di binding, consideriamo alcuni esempi del mondo reale. Un esempio di early binding è la procedura adottata da alcune Università anglosassoni, che consentono ai genitori di iscrivere un bambino alla nascita, e di pagare in anticipo le tasse di insegnamento, al prezzo corrente. Quando lo studente si presenta, 18 anni dopo, gli insegnamenti saranno già stati pagati completamente, anche se il loro costo in quel momento sarà più alto.

Nelle industrie manifatturiere, un esempio di early binding può essere quello di ordinare dei pezzi in anticipo e mantenere un loro magazzino. Al contrario, la produzione "just-in-time" richiede ai fornitori di essere in grado di fornire i componenti immediatamente, senza nessun preavviso, e questo è un esempio di late binding.

I linguaggi di programmazione spesso supportano diversi tempi di binding per le variabili. Le variabili globali sono associate dal compilatore ad un particolare indirizzo virtuale, e questo è un esempio di early binding; le variabili locali ad una procedura vengono assegnate ad un indirizzo virtuale (nello stack) nel momento in cui la procedura viene invocata, e questo è un esempio di binding intermedio. Le variabili memorizzate nel heap (quelle allocate con *malloc* in C o con *new* in Java) sono assegnate ad indirizzi virtuali solo nel momento in cui sono effettivamente utilizzate, e qui abbiamo un esempio di late binding.

Spesso i sistemi operativi utilizzano l'*early binding* per la maggior parte delle strutture dati, ma utilizzano il *late binding* occasionalmente, per ragioni di flessibilità, ed uno di questi casi è l'allocazione della memoria. I primi sistemi a multiprogrammazione presenti su macchine prive di hardware per la rilocazione degli indirizzi, dovevano caricare un programma in qualche indirizzo di memoria, e quindi rilocalarlo per farlo girare lì; se questo veniva portato fuori dalla memoria dallo swapper, doveva essere riportato allo stesso indirizzo di memoria altrimenti sarebbe fallito. Invece, la memoria virtuale paginata, è una forma di late binding, in quanto l'indirizzo fisico effettivo corrispondente ad un dato indirizzo virtuale, non è conosciuto finché la pagina non viene acceduta, ed inserita effettivamente in memoria.

Un altro esempio di late binding è la disposizione delle finestre in una GUI. A differenza dei primi sistemi grafici, in cui il programmatore doveva specificare le coordinate assolute dello schermo per tutte le immagini sullo schermo, nelle moderne GUI il software utilizza delle coordinate relative all'origine della finestra, che non è determinata finché la finestra non viene posta sullo schermo, e che successivamente può essere modificata.

## 12.3.6 Strutture statiche contro strutture dinamiche

I progettisti dei sistemi operativi sono costantemente obbligati a scegliere tra strutture statiche e strutture dinamiche. Le strutture statiche sono sempre le più semplici da capire, le più facili da programmare e le più veloci da utilizzare, mentre le strutture dinamiche sono più flessibili. Un esempio ovvio è dato dalla tabella dei processi: i vecchi sistemi, allocavano semplicemente un array di strutture, una per processo, di dimensione fissata; se la tabella dei processi era composta da 256 entrate, allora in ogni istante di tempo, potevano esistere solo 256 processi, e il tentativo di creare un 257-esimo processo sarebbe fallito per mancanza di spazio nella tabella. Considerazioni analoghe si possono fare per la tabella dei file aperti (sia quella di ogni utente, sia quella per tutto il sistema), e per molte altre tabelle del nucleo.

Una strategia alternativa è quella di costruire la tabella dei processi come una lista

```

trovato = 0;
for (p = &proc_table[0]; p<proc_tab|PROC_TABLE_SIZE]; p++) {
 if (p->proc_pid == pid) {
 trovato = 1;
 break;
 }
}

```

**Figura 12.4** Codice per la ricerca nella tavola del processo che ha un dato PID.

linkata di mini-tabelle, inizialmente solo una; dopodiché, se la tabella si riempie, ne viene allocata un'altra, prelevandola da un deposito globale di tabelle, e la si collega alla prima; in questo modo, la tabella dei processi non si satura fino a quando non si esaurisce tutta la memoria del nucleo.

D'altro canto, il codice per la ricerca nella tabella diventa più complicato, per esempio, in Figura 12.4 si può vedere il codice per ricercare, in una tabella di processi statica, se un dato PID, *pid*, è presente. Questa è semplice ed efficiente; mentre eseguire la stessa cosa su di una lista linkata di mini-tabelle richiede più lavoro.

Le tabelle statiche sono migliori quando la memoria è abbondante, oppure quando la dimensione utile può essere stimata accuratamente: per esempio, in un sistema a singolo utente, è improbabile che un utente esegua più di 32 processi nello stesso istante, e non capita un disastro totale se il tentativo di eseguire il 33-esimo processo fallisce.

Un'altra alternativa è quella di utilizzare una tabella di dimensione fissata, e se questa si riempie, allocare un'altra tabella di dimensione fissata, diciamo due volte più grande. Le entrate correnti sono quindi copiate nella tabella nuova, e la vecchia tabella ritorna nel deposito delle tabelle libere. In questo modo, la tabella è sempre contigua invece che collegata, ma in questo caso lo svantaggio è che sono necessari dei meccanismi per la gestione dei depositi delle tabelle, e che l'indirizzo della tabella in questo caso è una variabile invece di una costante.

Un problema simile avviene per gli stack del nucleo: quando un thread passa in modalità kernel, o si attiva un thread del nucleo, bisogna fornirgli uno stack nello spazio del nucleo. Per i thread dell'utente, lo stack può essere inizializzato affinché cresca verso il basso a partire dall'indirizzo più alto nello spazio di indirizzamento virtuale, in modo da non doverne specificare la dimensione in anticipo; mentre, per i thread del nucleo, la dimensione deve essere specificata in anticipo, poiché lo stack porta via un po' di spazio di indirizzi del nucleo, e ci possono essere più stack. La questione è: quanto spazio dovrebbe avere ognuno? I compromessi, qui, sono simili a quelli per la tabella dei processi.

Un altro compromesso tra statico e dinamico riguarda la schedulazione dei processi. In alcuni sistemi, specialmente quelli in tempo reale, la schedulazione può essere effettuata staticamente in anticipo; per esempio, una compagnia aerea conosce gli orari di partenza dei suoi voli settimane prima della loro partenza. In modo simile, i sistemi multimediali conoscono in anticipo quando schedolare l'audio, il video, e altri processi, mentre per utilizzi general-purpose, queste considerazioni non reggono e la schedulazione deve essere dinamica.

Un altro problema statico-dinamico è la struttura del nucleo. Se il nucleo è collegato come un singolo programma binario e, per girare, è caricato nella memoria, tutto risulta più semplice; tuttavia, la conseguenza di questa scelta è che aggiungere un nuovo dispositivo di I/O richiede un ulteriore collegamento del nucleo con il nuovo driver del dispositivo. Le vecchie versioni di UNIX seguivano questa strada, ed erano abbastanza soddi-

sfacenti in un ambiente di minicomputer, quando l'aggiunta di un nuovo dispositivo di I/O era un avvenimento raro; oggi invece molti sistemi operativi consentono l'aggiunta di codice nel nucleo in maniera dinamica, con tutta la complessità aggiuntiva che questo comporta.

### 12.3.7 Implementazione top-down contro implementazione bottom-up

Mentre è molto meglio progettare un sistema in modo top-down, in teoria si può implementarlo in modo top-down oppure in modo bottom-up. In una implementazione top-down, gli implementatori incominciano dal gestore delle chiamate di sistema, e guardano quali meccanismi e quali strutture sono necessarie per supportare tali chiamate, poi scrivono queste procedure, e via di seguito, fino a raggiungere l'hardware.

Il problema, con questo approccio, è che è difficile collaudare qualcosa, avendo a disposizione solo le procedure ad alto livello; per questa ragione, attualmente, molti sviluppatori trovano più pratico costruire il sistema in modo bottom-up. Questo approccio implica, per prima cosa, scrivere il codice che nasconde l'hardware di basso livello, essenzialmente l'HAL in Figura 11.7 e, molto presto, diventa necessario avere anche la gestione delle interruzioni e il driver del clock.

Quindi si passa ad affrontare la multiprogrammazione, con un semplice schedulatore (per esempio, con la schedulazione round-robin); a questo punto si dovrebbe poter collaudare il sistema per vedere se possono girare correttamente molti processi; se questo funziona, è il momento di definire accuratamente le varie tabelle e le strutture dati necessarie all'interno del sistema, specialmente quelle per la gestione dei processi e dei thread, e successivamente quelle per la gestione della memoria. Inizialmente l'I/O ed il file system possono aspettare, ad eccezione di un modo primitivo di lettura della tastiera e di scrittura sullo schermo, per le operazioni di test e di debug. In alcuni casi, le strutture chiave di basso livello dovrebbero essere protette, consentendo l'accesso soltanto attraverso procedure di accesso specifiche – in effetti, questa è programmazione orientata agli oggetti, a prescindere dal linguaggio di programmazione utilizzato. I livelli bassi, una volta completati, possono essere accuratamente collaudati, e in questo modo, il sistema avanza dal basso verso l'alto, all'incirca nel modo in cui gli imprenditori edili costruiscono grattacieli per uffici.

Se è disponibile un gruppo di persone numeroso, un approccio alternativo è quello di completare, per prima cosa, il progetto dettagliato dell'intero sistema, e quindi assegnare gruppi differenti a scrivere moduli differenti, dove ogni gruppo controlla il proprio lavoro, isolato dagli altri; quando tutti i pezzi sono pronti, vengono integrati e controllati. Il problema, con questa linea di attacco, è che, se inizialmente non funziona nulla, può essere difficile isolare se il problema sia che uno o più moduli funzionano male, o se un gruppo ha frainteso cosa doveva fare qualche altro modulo; tuttavia, con grandi gruppi di lavoro, spesso si usa questo approccio per massimizzare il livello di parallelismo nel lavoro di programmazione.

### 12.3.8 Tecniche utili

Abbiamo appena esaminato alcune idee astratte per la progettazione e l'implementazione di un sistema; ora esamineremo alcune tecniche concrete utili per l'implementazione di un sistema. Ve ne sono molte altre, logicamente, ma le limitazioni di spazio ci impongono di esaminarne solo alcune.

## Nascondere l'hardware

La maggior parte dell'hardware è brutto, e deve essere nascosto nei primi livelli (a meno di non volerne esporre la potenza, cosa che la maggior parte dell'hardware non ha). Alcuni dettagli di livello molto basso possono essere nascosti da uno strato di tipo HAL mostrato in Figura 12.2; tuttavia, molti dettagli hardware non possono essere nascosti in questo modo.

Una cosa che merita attenzione da subito è la gestione delle interruzioni, infatti le interruzioni rendono sgradevole la programmazione, ma i sistemi operativi devono affrontarle. Un approccio è quello di convertirle immediatamente in qualcosa' altro: per esempio, ogni interruzione potrebbe essere convertita istantaneamente in un thread che si attiva improvvisamente (pop-up), a quel punto si programmerà con i thread e non con le interruzioni.

Un secondo approccio è quello di convertire ogni interruzione in una operazione di unlock su un mutex su cui sta aspettando il driver corrispondente; in questo caso l'unico effetto di una interruzione è quello di far diventare ready qualche thread.

Un terzo approccio è quello di convertire una interruzione in un messaggio per qualche thread: il codice a basso livello costruisce semplicemente un messaggio, che dice da dove viene l'interruzione, lo mette nella coda, e chiama lo scheduler per (potenzialmente) far partire il gestore, che, probabilmente, era bloccato in attesa di quel messaggio. Tutte queste tecniche, e altre simili, cercano di convertire le interruzioni in operazioni di sincronizzazione dei thread, in quanto, se ogni interruzione è gestita da un thread adatto, in un contesto adatto, è più facile da gestire rispetto ad avere un handler (gestore) eseguito nel contesto arbitrario in cui, casualmente, si è presentato. Naturalmente, questo si deve fare in maniera efficiente, ma negli strati interni del sistema operativo, tutto deve essere fatto in maniera efficiente.

La maggior parte dei sistemi operativi è progettata per girare su diverse piattaforme hardware, che possono differire come chip di CPU, o di MMU, come lunghezza delle parole, come dimensione della RAM, e in altre caratteristiche che non possono essere facilmente mascherate dall'HAL o da qualcosa di equivalente. Tuttavia, è molto desiderabile avere un solo insieme di file sorgente da utilizzare per generare tutte le versioni, altrimenti ogni baco che si presenta successivamente deve essere corretto diverse volte in diversi sorgenti, con il pericolo che tali sorgenti si allontanino l'uno dall'altro.

Alcune differenze dell'hardware, come ad esempio la dimensione della RAM, possono essere trattate dal sistema operativo, determinando a tempo di boot il valore relativo, e conservandolo in una variabile. Gli allocator di memoria, per esempio, possono utilizzare una variabile, contenente la dimensione della RAM, per determinare quanto grande vada creata la cache dei blocchi, le tabelle delle pagine, eccetera; anche le tabelle statiche, come la tabella dei processi, possono essere dimensionate in base alla memoria totale disponibile.

Tuttavia, altre differenze, come ad esempio avere diversi chip di CPU, non si possono risolvere con un solo programma binario, che determina a tempo di esecuzione su quale CPU sta lavorando. Un modo per affrontare il problema di avere un solo sorgente e diversi codici oggetto è quello di utilizzare la compilazione condizionale: nei file sorgente si definiscono, a tempo di compilazione, alcuni parametri per configurazioni differenti, e questi si utilizzano per individuare le parti di codice dipendenti dalla CPU, dalla lunghezza della parola, dalla MMU, eccetera. Per esempio, si immagini un sistema operativo che deve girare su un chip Pentium e su un chip UltraSPARC, i quali necessitano di un codice di inizializzazione diverso; la procedura *init* potrebbe essere scritta come illustrato in Figura 12.5(a); a seconda del valore del parametro *CPU*, che è definito nel file di intestazione *config.h*, si realizza un tipo o l'altro di inizializzazione. In questo modo non

```

#include "config.h"
init()
{
#if (CPU == PENTIUM)
/* Inizializzazione del Pentium qui*/
#endif

#if (CPU == ULTRASPARC)
/* inizializzazione dell'UltraSPARC qui.*/
#endif
}

#include "config.h"
#if (WORD_LENGTH == 32)
typedef int Registro;
#endif

#if (WORD_LENGTH == 64)
typedef long Registro;
#endif

Registro R0, R1, R2, R3;

```

(a)

(b)

**Figura 12.5** [a] Compilazione dipendente dalla CPU. [b] Compilazione condizionale dipendente dalla lunghezza delle parole.

vi sono perdite di efficienza, in quanto il programma binario contiene solo il codice necessario per la macchina che interessa.

Come secondo esempio, supponiamo ci sia la necessità di avere un tipo di dato *Registro*, che dovrebbe essere di 32 bit su un Pentium e di 64 bit su un UltraSPARC; questo si potrebbe gestire con il codice condizionale di Figura 12.5(b) (supponendo che il compilatore produca *int* di 32 bit e *long* di 64 bit). Una volta data questa definizione (probabilmente in un file di intestazione incluso ovunque), il programmatore può limitarsi a dichiarare variabili di tipo *Registro*, sapendo che avranno la lunghezza giusta.

Logicamente il file di intestazione, *config.h*, deve essere definito correttamente: ad esempio per il Pentium potrebbe essere qualcosa come:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

Per compilare il sistema per un'UltraSPARC, si utilizzerebbe un *config.h* diverso, con valori corretti per l'UltraSPARC, probabilmente qualcosa come:

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Alcuni lettori potrebbero chiedersi perché CPU e WORD\_LENGTH sono gestiti da macro diverse, in quanto avremmo potuto facilmente parametrizzare la definizione di *Registro* con un test sulla CPU, impostandolo a 32 bit per il Pentium e a 64 bit per l'UltraSPARC; ma questa non sarebbe una buona idea. Si consideri cosa accadrebbe se in seguito portassimo il sistema su un Intel Alpha a 64 bit: dovremmo aggiungere una terza condizione alla Figura 12.5(b) per l'Alpha. Al contrario, per realizzare ciò che abbiamo detto, tutto quello che dobbiamo fare è includere la linea

```
#define WORD_LENGTH 64
```

nel file *config.h* per l'Alpha.

Questo esempio illustra il principio di ortogonalità che abbiamo discusso precedentemente. Questi elementi, che sono dipendenti dalla CPU, dovrebbero essere compilati sotto condizione in base alla macro CPU, e tutto ciò che dipende dalla lunghezza delle par-

le dovrebbe utilizzare la macro WORD\_LENGTH. Si possono fare considerazioni simili per molti altri parametri.

## Indirezione

Alcune volte si dice che non ci siano problemi nell'informatica, che non si possano risolvere con un altro livello di indirezione: insieme con un po' di esagerazione, in questa affermazione c'è senza alcun dubbio un fondo di verità. Consideriamo alcuni esempi: sui sistemi Pentium, quando viene premuto un tasto, l'hardware genera una interruzione, ed inserisce il numero del tasto, invece del suo codice ASCII, in un registro di dispositivo. Successivamente, quando più tardi si rilascia il tasto, si genera una seconda interruzione, ed anch'essa memorizza il numero del tasto. Questa indirezione consente al sistema operativo di usare il numero del tasto come indice in una tabella contenente il carattere ASCII corrispondente, e questo rende facile gestire le numerose tastiere diverse, utilizzate nei diversi paesi del mondo. Considerare informazioni sia sulla pressione sia sul rilascio dei tasti, rende possibile utilizzare ogni tasto come un tasto delle maiuscole (tasto shift), poiché il sistema operativo conosce la sequenza esatta con cui i tasti vengono schiacciati e rilasciati.

L'indirezione è utilizzata anche per l'output: i programmi possono scrivere caratteri ASCII sullo schermo, ma questi sono interpretati come indici nella tabella del font di output corrente, e l'entrata nella tabella contiene la mappa di bit del carattere. Questa indirezione rende possibile separare i caratteri dai font.

Un altro esempio di indirezione è l'utilizzo, in UNIX, dei *major device number* (numeri principali dei dispositivi). All'interno del nucleo c'è una tabella indicizzata con il major device number del dispositivo per i dispositivi a blocchi, e un'altra per i dispositivi a caratteri. Quando un processo apre un file speciale come ad esempio /dev/hd0, il sistema estrae dall'i-node il tipo (blocco o carattere) e i major e minor device number, di cui il primo costituisce l'indice per trovare il driver, nella tabella dei driver appropriata. Questa indirezione rende facilmente possibile la reconfigurazione del sistema, poiché i programmi hanno a che fare con i nomi simbolici dei dispositivi, non con i nomi dei driver effettivi.

Un altro esempio di indirezione avviene nei sistemi a scambio di messaggi, che indicano come destinatario del messaggio una casella di posta invece di un processo: attraverso l'indirezione tramite caselle di posta (invece di nominare il processo destinazione), si può raggiungere una notevole flessibilità (ad esempio, è come avere una segretaria che gestisce i messaggi del suo capo).

In un certo senso, l'utilizzo di macro, come ad esempio

```
#define PROC_TABLE_SIZE 256
```

è anch'essa una forma di indirezione, poiché il programmatore può scrivere il codice senza dover conoscere quanto deve essere realmente grande la tabella. È una buona abitudine dare nomi simbolici a tutte le costanti (ad eccezione qualche volta di -1, 0 e 1), e metterli nelle intestazioni, con alcuni commenti per spiegare a cosa servono.

## Riusabilità

Molto spesso è possibile riutilizzare lo stesso codice in contesti leggermente differenti. Questa è una buon'idea poiché riduce la dimensione del programma binario, e perché il codice viene controllato una sola volta: per esempio, si supponga di usare le mappe di bit per tenere traccia dei blocchi liberi sul disco: la gestione dei blocchi sul disco può essere affidata alle procedure *alloc* e *free* che gestiscono le mappe di bit.

Come minimo, queste procedure dovrebbero funzionare per ogni disco; ma possiamo andare oltre, perché le stesse procedure possono anche funzionare per la gestione dei blocchi di memoria, dei blocchi nella cache del file system, e degli i-node; infatti, si possono utilizzare per allocare e deallocare ogni risorsa che può essere numerata linearmente.

## Rientranza

La rientranza si riferisce alla capacità del codice di essere eseguito due o più volte simultaneamente. Infatti su un multiprocessore, c'è sempre il pericolo che, mentre una CPU sta eseguendo qualche procedura, un'altra CPU cominci ad eseguire anch'essa la stessa procedura, prima che la prima CPU abbia finito; in questo caso, due (o più) thread potrebbero eseguire lo stesso codice, nello stesso momento, su differenti CPU. Questa situazione deve essere protetta, utilizzando i mutex o qualche altro mezzo per proteggere le regioni critiche.

Tuttavia, il problema esiste anche su un singolo processore, in particolare, per la maggior parte del tempo ogni sistema operativo è in esecuzione con le interruzioni abilitate (se facesse altrimenti, perderebbe molte interruzioni e diventerebbe inaffidabile), e mentre il sistema operativo è occupato nell'esecuzione di una qualche procedura,  $P$ , è assolutamente possibile che capiti una interruzione, e che il gestore delle interruzioni chiami anch'esso  $P$ . Se le strutture dati di  $P$  fossero in uno stato inconsistente al momento dell'interruzione, il gestore le vedrà in uno stato inconsistente e fallirà.

Un esempio ovvio in cui questo può accadere è il caso in cui  $P$  è lo scheduler: si supponga che un qualche processo abbia terminato il suo quanto di tempo, e che il sistema operativo lo stia spostando alla fine della coda. Se, durante la manipolazione della lista, avviene un'interruzione, alcuni processi potrebbero diventare ready e si fa partire lo scheduler: con le code in uno stato inconsistente, il sistema probabilmente crollerà. Di conseguenza si può affermare che, anche in un monoprocesso, è meglio che la maggior parte del sistema operativo sia rientrante, che le strutture dati critiche siano protette dai mutex, e che le interruzioni siano disabilitate nei momenti in cui non possono essere accettate.

## Forza bruta

Il criterio di utilizzare la forza bruta per risolvere i problemi ha preso una cattiva fama nel corso degli anni, ma spesso è la strada migliore da seguire, in termini di semplicità. Ogni sistema operativo ha molte procedure che sono chiamate raramente, oppure che operano con così pochi dati che non vale la pena ottimizzarle. Per esempio, spesso è necessario fare ricerche in varie tabelle ed array all'interno del sistema. L'algoritmo della forza bruta implica lasciare i dati nella tabella nell'ordine in cui sono stati inseriti, e scorrere linearmente la tabella nel momento in cui si deve cercare qualcosa; se il numero dei dati è piccolo (ad esempio sotto 100), il guadagno che si ottiene nell'ordinare o nel gestire con un codice hash la tabella è piccolo, ma il codice è molto più complicato ed è più probabile che conteriga bachi.

Certamente, per le funzioni che sono nel cammino critico, ad esempio, nel cambio di contesto, si dovrebbe fare di tutto per renderle veloci, eventualmente anche scrivendole (il Cielo non voglia!) in linguaggio assembler. Ma molte parti del sistema non si trovano nel cammino critico; per esempio, molte chiamate di sistema sono attivate raramente. Se c'è una fork ogni 10 s, ed essa richiede 10 ms per essere completata, allora anche ottimizzandola a 0, si guadagnerebbe solo lo 0,1%. Se il codice ottimizzato è più lungo e contiene più bachi, allora non è il caso di preoccuparsi dell'ottimizzazione.

## Anticipare il controllo di errori

Molte chiamate di sistema possono, potenzialmente, fallire per una varietà di ragioni: il file da aprire appartiene a qualcun altro; la creazione dei processi fallisce in quanto la tabella dei processi è piena; oppure, non si può spedire un segnale perché il processo destinatario non esiste, quindi, il sistema operativo deve controllare accuratamente ogni possibile errore, prima di effettuare la chiamata.

Molte chiamate di sistema richiedono anche l'acquisizione di risorse come ad esempio slot della tabella dei processi, slot della tabella degli i-node, o descrittori di file. Un consiglio generale che evita molti dolori, è quello di controllare, prima di tutto, se la chiamata di sistema può essere realmente effettuata prima di acquisire qualsiasi risorsa. Questo significa mettere tutti i controlli all'inizio della procedura che esegue la chiamata di sistema, ed ogni controllo dovrebbe essere della forma:

```
If (condizione_di_errore) return (CODICE_ERRORE)
```

Se la chiamata riesce a superare positivamente tutti i controlli, allora di sicuro avrà successo, e a quel punto si possono acquisire le risorse.

Inframmezzare i controlli con l'acquisizione delle risorse significa che, se qualche controllo fallisce lungo la strada, tutte le risorse acquisite fino a quel momento devono essere restituite. Se si sbaglia a questo punto, e non si restituiscono alcune risorse, potrebbe non esserci nessun danno immediato, ma, per esempio, un'entrata nella tabella dei processi potrebbe diventare permanentemente non disponibile. Tuttavia, in un periodo di tempo più lungo, questo baco può verificarsi diverse volte; e prima o poi, la maggior parte delle entrate della tabella dei processi (o anche tutte) potrebbero diventare non disponibili, causando un crollo del sistema in modo veramente imprevedibile e difficile da eliminare.

Molti sistemi presentano questo problema sotto forma di perdite di memoria. Tipicamente, il programma chiama la *malloc* per allocare spazio, ma dimentica, successivamente, di chiamare la *free* per rilasciarlo, e così, gradualmente, tutta la memoria scompare, e bisogna far ripartire il sistema.

Engler ed altri (2000) hanno proposto un modo interessante per controllare alcuni di questi errori a tempo di compilazione; essi hanno osservato che il programmatore conosce molte invarianti che il compilatore non conosce. Ad esempio, dopo aver fatto *lock* su un mutex, tutti i percorsi che partono dall'istruzione di *lock* devono contenere una *unlock*, e nessun'altra *lock* sullo stesso mutex. Essi hanno escogitato un modo in cui il programmatore riesce a dire questo fatto al compilatore, istruendolo a controllare, a tempo di compilazione, se ci sono violazioni dell'invariante in qualche percorso. Inoltre, il programmatore può anche specificare di controllare che la memoria allocata sia rilasciata in tutti i percorsi possibili, e molte altre condizioni.

## 12.4 Prestazioni

A parità di tutto il resto, un sistema operativo veloce è migliore di uno lento; tuttavia, un sistema operativo veloce ma inaffidabile non vale quanto un sistema operativo lento ma affidabile. Poiché le ottimizzazioni complesse spesso portano ad avere bachi, è importante utilizzarle con moderazione; nonostante ciò, vi sono casi in cui le prestazioni sono critiche, e dove quindi vale la pena di ottimizzare. Nelle sezioni seguenti osserveremo alcune tecniche generali che possono essere utilizzate, laddove siano richieste, per migliorare le prestazioni.

## 12.4.1 Perché i sistemi operativi sono lenti?

Prima di parlare delle tecniche di ottimizzazione, vale la pena precisare che la lentezza di molti sistemi operativi è in larga misura causata da loro stessi: per esempio, i vecchi sistemi operativi, come MS-DOS e la versione 7 di UNIX, si avviavano in pochi secondi, mentre i sistemi UNIX moderni e Windows 2000 possono impiegare alcuni minuti per avviarsi, anche se stanno girando su hardware che è 100 volte più veloce. La ragione è che i sistemi stanno facendo molto di più, che l'utente lo voglia o meno. Eccovi subito un esempio: Plug and Play rende un po' più facile installare un nuovo dispositivo hardware, ma il prezzo pagato è che ad ogni avvio, il sistema operativo deve andare ad ispezionare tutto l'hardware per vedere se, lì fuori, c'è qualcosa di nuovo, e questa scansione dei bus richiede del tempo.

Un approccio alternativo (migliore, secondo l'autore) sarebbe quello di accantonare il plug-and-play ed avere un'icona nello schermo etichettata "Installa nuovo hardware". Per installare una nuova componente hardware, l'utente inizia la scansione dei bus cliccando su questa icona, invece di scadirli ad ogni avvio del sistema. I progettisti dei sistemi attuali, erano di sicuro a conoscenza di questa opzione, ma la rifiutarono, sostanzialmente, perché supponevano che gli utenti fossero troppo stupidi per poter fare tutto questo correttamente (sebbene si vogliano esprimere in modo più gentile). Questo è solo un esempio, ma vi sono molti altri casi in cui il desiderio di rendere il sistema "user-friendly" (o "a prova di idiota", a seconda del punto di vista), rallenta il sistema tutte le volte per tutti gli utenti.

Probabilmente la cosa migliore che i progettisti del sistema possono fare per migliorare le prestazioni è di essere molto più selettivi nell'aggiungere nuove caratteristiche; la domanda da fare non è: "Questo piacerà all'utente?" ma "Vale la pena realizzare questa caratteristica al prezzo inevitabile della dimensione del codice, della velocità, della complessità e dell'affidabilità?"; la nuova caratteristica dovrebbe essere inclusa solo se i vantaggi superano chiaramente gli svantaggi. I programmati hanno la tendenza a supporre che la dimensione del codice e il numero dei bachi saranno nulli, e che la velocità sarà infinita, ma le esperienze dimostrano che questa visione è un po' ottimistica.

Un altro fattore che gioca un ruolo importante è il marketing dei prodotti; nel momento in cui la versione 4 o 5 di un dato prodotto raggiunge il mercato, probabilmente tutte le caratteristiche che sono davvero utili sono già state incluse, e la maggior parte delle persone che hanno bisogno del prodotto lo possiede già. Nonostante ciò, per incrementare le vendite, molti produttori continuano a produrre nuove versioni, con più caratteristiche, in modo da poter vendere ai loro clienti le versioni aggiornate: aggiungere nuove caratteristiche solo per il gusto di aggiungere nuove caratteristiche, può aiutare le vendite ma raramente aiuta le prestazioni.

## 12.4.2 Che cosa dovrebbe essere ottimizzato?

Come regola generale, la prima versione del sistema dovrebbe essere la più semplice possibile. Le uniche ottimizzazioni dovrebbero riguardare tutte quelle cose che sono talmente ovvie che diventa un problema non averle, ad esempio avere la cache dei blocchi nel file system. Una volta che il sistema è installato e sta girando, si dovrebbero fare delle misurazioni per vedere dove *realmente* il tempo viene speso: sulla base di questi valori, si dovrebbero fare le ottimizzazioni là dove servono maggiormente.

Ecco una storia vera in cui un'ottimizzazione ha dato più danni che vantaggi: uno degli studenti dell'autore (che preferisce non essere nominato) scrisse il programma *mks* di MINIX, che inseriva un nuovo file system in un disco appena formattato. Lo studente

spese circa 6 mesi per ottimizzarlo, includendo l'inserimento di una cache nel disco; quando lo finì, il programma non funzionava e impiegò altri mesi per il debug. Questo programma tipicamente gira, in tutta la vita di un computer, una sola volta sull'hard disk, quando s'installa il sistema, ed inoltre gira una volta per ogni floppy disk formattato, ed ogni sua esecuzione impiega circa 2 secondi. Quindi, anche se la versione non ottimizzata avesse richiesto 1 minuto, il programma utilizzava così poche risorse che non valeva la pena spendere troppo tempo ad ottimizzarlo, dato che non veniva utilizzato frequentemente.

Uno slogan che si applica molto bene all'ottimizzazione delle prestazioni è:

*Quello che è abbastanza buono è abbastanza buono.*

Con questo affermiamo che quando le prestazioni hanno raggiunto un livello ragionevole, probabilmente non vale la pena eseguire altro lavoro e aumentare la complessità per spremere un piccolo miglioramento. Se l'algoritmo di schedulazione è ragionevolmente equo, e la CPU risulta occupata per il 90% del tempo, allora sta facendo bene il suo lavoro, e progettarne una versione molto più complessa che è migliore del 5%, probabilmente non è una buona idea. In modo simile, se il tasso di page fault è abbastanza basso da non costituire un collo di bottiglia, non vale la pena fare peripezie per ottenere prestazioni ottimali. Evitare disastri è molto più importante che ottenere prestazioni ottimali, specialmente dato che ciò che è ottimo con un certo carico di lavoro può non essere ottimo con un altro.

### 12.4.3 Compromessi tra spazio e tempo

Un approccio generale per migliorare le prestazioni è quello di effettuare dei compromessi tra lo spazio ed il tempo; nell'informatica accade frequentemente che si possa scegliere tra un algoritmo che utilizza poca memoria, ma che risulta lento, ed un algoritmo che utilizza molta più memoria, ma che risulta più veloce. Quando si effettua una ottimizzazione importante, vale la pena tenere in considerazione gli algoritmi che guadagnano in velocità utilizzando più memoria, o viceversa, che salvano memoria preziosa eseguendo più calcoli.

Una tecnica che è spesso utile, è quella di sostituire le piccole procedure con delle macro: utilizzare una macro elimina la perdita di tempo (overhead) normalmente associata ad una chiamata di procedura; il guadagno è particolarmente significativo se la chiamata avviene all'interno di un ciclo. Ad esempio, supponiamo di utilizzare delle mappe di bit per tenere traccia delle risorse, e di avere spesso bisogno di conoscere quante unità sono libere in alcune porzioni delle mappe di bit; per realizzare questo obiettivo, ci serve una procedura, *num\_bit*, che conta il numero dei bit a 1 in un byte. La procedura completa è data in Figura 12.6(a), ed effettua un ciclo sui bit di un byte contandoli uno alla volta.

Questa procedura ha due fonti di inefficienza: primo, deve essere chiamata, allocando spazio nello stack per la procedura stessa, e deve ritornare: ogni chiamata di procedura ha un tale overhead; secondo, contiene un ciclo, e c'è sempre un overhead associato ad ogni ciclo.

Un approccio completamente differente è quello di utilizzare la macro di Figura 12.6(b): è una espressione in linea che calcola la somma dei bit attraverso spostamenti (shift) successivi dell'argomento, mascherando tutto tranne il bit più basso, e sommando gli otto termini: la macro non è sicuramente un'opera d'arte, ma compare nel codice solo una volta. Quando la macro viene chiamata, per esempio, con

```
#define DIM_BYTE 8 /* Un byte contiene 8 bit */

int num_bit(int byte)
{
 int i, num = 0;
 for (i = 0; i < DIM_BYTE; i++) /* Conteggio dei bit in un byte */
 if ((byte >> i) & 1) num++; /* Ciclo sui bit di un byte */
 return(num); /* Se il bit è 1, si incrementa num */
} /* Si restituisce la somma */

(a)
```

```
/ * Macro che aggiunge i bit in un byte e restituisce la somma *
#define num_bit(b) ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + ((b>>4)&1) +
 ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)

(b)
```

```
/ * Macro che cerca il numero dei bit in una tabella *
char bit[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define num_bit(b) (int) bit[b]
```

(c)

**Figura 12.6** [a] Una procedura per contare i bit di un byte. [b] Una macro per contare i bit. [c] Una macro che cerca il totale dei bit in una tabella.

```
Somma = num_bit(table[i]);
```

la chiamata della macro sembra identica alla chiamata della procedura, di conseguenza il codice, nel caso della macro, a parte una definizione alquanto confusa, non presenta nessun peggioramento rispetto al caso della procedura, ma è molto più efficiente poiché elimina sia l'overhead della chiamata di procedura sia l'overhead del ciclo.

Possiamo portare questo esempio un passo avanti. Perché effettuare il conteggio dei bit? Perché non cercarlo in una tabella? Dopo tutto, ci sono solo 256 byte diversi, ognuno con un unico valore tra 0 ed 8; possiamo dichiarare una tabella di 256 entrate, bit, inizializzando ogni entrata (a tempo di compilazione) con il numero del bit corrispondente al valore del byte rispettivo, e con questo approccio non è necessario nessun calcolo a tempo di esecuzione, ma solo una operazione di indicizzazione. La macro che fa il lavoro è data in Figura 12.6(c).

Questo è un chiaro esempio di utilizzo del tempo di calcolo invece della memoria, tuttavia, potremmo andare ancora oltre. Se sono necessari i conteggi dei bit per le parole di 32 bit, utilizzando la nostra macro *num\_bit*, abbiamo bisogno di eseguire quattro controlli per ogni parola; ma se espandiamo la tabella a 65536 entrate, bastano due controlli per parola, al prezzo di una tabella molto più grande.

Le tabelle possono essere utilizzate in altri modi, per esempio, nel Capitolo 7 abbiamo visto come viene effettuata una compressione di una immagine JPEG, con complesse trasformazioni discrete di coseni; una tecnica alternativa di compressione, GIF, utilizza una tabella di ricerca, per codificare i pixel RGB a 24 bit. Tuttavia, GIF lavora solo su imma-

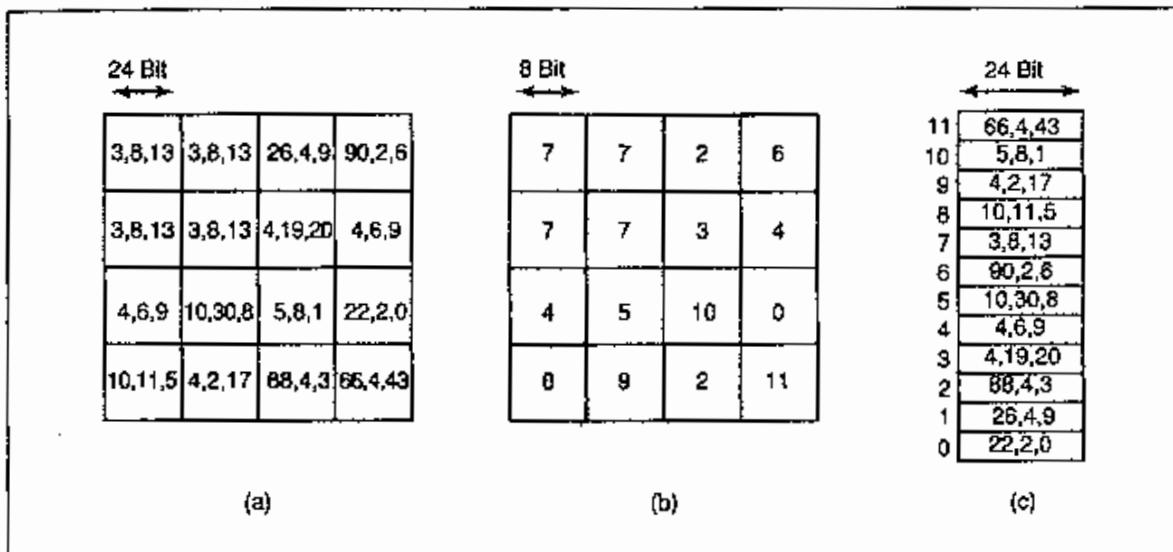


Figura 12.7 (a) Parte di una immagine non compressa con 24 bit per pixel. (b) La stessa parte compressa con GIF, con 8 bit per pixel. (c) La tavolozza dei colori.

gini con 256 colori, o meno, e per ogni immagine da comprimere, si costruisce una tavolozza con 256 ingressi, ciascuno contenente una valore RGB a 24 bit. L'immagine compressa contiene quindi un indice di 8 bit per ogni pixel, invece di un valore di colore di 24 bit, con un guadagno di un fattore tre. Illustriamo questa idea nella Figura 12.7: per una sezione 4x4 di un'immagine, l'immagine originale non compressa è mostrata in Figura 12.7(a), dove ogni valore è un valore di 24 bit, in cui ogni gruppo di 8 bit fornisce l'intensità per il rosso, il verde ed il blu. L'immagine GIF è mostrata in Figura 12.7(b), dove ogni valore è un indice ad 8 bit nella tavolozza dei colori; quest'ultima è memorizzata nel file dell'immagine, ed è mostrata in Figura 12.7(c). In effetti, c'è dell'altro nelle GIF, ma il fulcro dell'idea è la tabella di ricerca.

C'è un altro modo per ridurre la dimensione di una immagine, che evidenzia un compromesso differente: PostScript è un linguaggio di programmazione che può essere utilizzato per descrivere immagini (in effetti, ogni linguaggio di programmazione può descrivere immagini, ma PostScript è adatto per questo scopo). Molte stampanti hanno un interprete PostScript inserito all'interno, per consentire l'esecuzione dei programmi PostScript ad esse inviati.

Per esempio, se in un'immagine c'è un blocco rettangolare di pixel dello stesso colore, un programma PostScript per tale immagine conterrà le istruzioni per piazzare un rettangolo in una certa posizione, e per riempirlo di un certo colore; per specificare questo comando è necessaria soltanto una manciata di bit. Quando l'immagine viene ricevuta dalla stampante, un interprete presente in essa, deve eseguire il programma per costruire l'immagine; quindi PostScript realizza la compressione dei dati al costo di maggiori calcoli, con un compromesso differente da quello della tabella di ricerca, ma altrettanto prezioso soprattutto quando scarseggia la memoria o la larghezza di banda.

Altri compromessi spesso coinvolgono le strutture dati. Le liste linkate doppiamente richiedono più memoria delle liste linkate semplici, ma spesso consentono un accesso più veloce agli elementi; le tabelle hash sprecano ancora più spazio, ma sono ancora più veloci. In breve, una delle cose principali da considerare quando si ottimizza un pezzo di codice è vedere se l'utilizzo di diverse strutture dati migliora il compromesso tra spazio e tempo.

## 12.4.4 L'utilizzo della cache

Una tecnica molto conosciuta per migliorare le prestazioni è l'utilizzo della cache applicabile ogni volta in cui lo stesso risultato sarà necessario diverse volte. L'approccio generale è quello di eseguire tutto il lavoro la prima volta, e quindi salvare il risultato in una cache; per gli utilizzzi successivi, la cache si controlla per prima, quindi, se il risultato è presente, lo si riutilizza, altrimenti, viene rieseguito l'intero lavoro.

Abbiamo già visto l'utilizzo della cache all'interno del file system, per mantenere alcuni blocchi di disco utilizzati di recente, risparmiando una lettura da disco ogni volta che si trova l'informazione (hit), tuttavia, la cache può essere utilizzata altrettanto bene per molti altri scopi; ad esempio, l'analisi sintattica (parsing) dei nomi di percorso è sorprendentemente costosa. Si consideri nuovamente l'esempio di UNIX in Figura 6.39: controllare il percorso `/usr/ast/mbox` richiede i seguenti accessi su disco:

1. Leggere l'i-node della directory radice (i-node 1).
2. Leggere la directory radice (blocco 1).
3. Leggere l'i-node di `/usr` (i-node 6).
4. Leggere la directory `/usr` (blocco 132).
5. Leggere l'i-node di `/usr/ast` (i-node 26).
6. Leggere la directory `/usr/ast` (blocco 406).

In questo esempio si effettuano 6 accessi su disco solo per scoprire il numero di i-node del file, quindi bisogna leggere tale i-node per scoprire i numeri dei blocchi su disco; se il file è più piccolo della dimensione del blocco (ad esempio 1024 byte), per leggere i dati si effettuano 8 accessi su disco.

Alcuni sistemi ottimizzano il parsing dei nomi di percorso mettendo in una cache le coppie (percorso, i-node): seguendo l'esempio di Figura 6.39, la cache, dopo aver effettuato il parsing di `/usr/ast/mbox`, contiene sicuramente le prime 3 entrate di Figura 12.8, mentre le ultime tre entrate si ottengono dal parsing degli altri percorsi.

Quando si deve controllare un percorso, l'analizzatore sintattico dei nomi (parser) controlla prima di tutto la cache, e ricerca la sottostringa più lunga presente in essa; per esempio, se si cerca il percorso `/usr/ast/fondi/stw`, la cache restituisce il fatto che `/usr/ast` è l'i-node 26, così la ricerca può iniziare da lì, eliminando quattro accessi su disco.

Un problema presente nell'utilizzare la cache per i percorsi è che la mappatura tra il nome del file ed il numero di i-node non è fissa per tutto il tempo. Si supponga che il file `/usr/ast/mbox` sia rimosso dal sistema ed il suo i-node venga riutilizzato per un file differente posseduto da un utente differente; in seguito, si crea nuovamente il file

| Percorso                       | Numero dell'i-node |
|--------------------------------|--------------------|
| <code>/usr</code>              | 6                  |
| <code>/usr/ast</code>          | 26                 |
| <code>/usr/ast/mbox</code>     | 60                 |
| <code>/usr/ast/libro</code>    | 92                 |
| <code>/usr/bal</code>          | 45                 |
| <code>/usr/bal/paper.ps</code> | 85                 |

Figura 12.8 Parte della cache degli i-node per la Figura 6.39.

`/usr/fast/mbox`, e questa volta prende l'i-node 106: se non viene fatto nulla per impedirlo, l'entrata della cache ora sarà errata, e le ricerche successive restituiranno il numero di i-node sbagliato. Per questo motivo, quando un file o una directory viene cancellata, l'entrata della cache corrispondente (e, se è una directory, tutte le entrate sotto di lei) dovranno essere eliminate dalla cache stessa.

I blocchi su disco ed i nomi di percorso non sono gli unici elementi che possono essere inseriti nella cache: si possono inserire anche gli i-node, e inoltre, se per gestire le interruzioni sono usati i thread di pop-up, ciascuno dei quali richiede uno stack ed alcuni meccanismi aggiuntivi, anche i thread precedentemente utilizzati possono essere inseriti nella cache, in quanto riutilizzare un thread usato è più semplice che creare uno nuovo da zero (per evitare di rallocare memoria). Praticamente tutto quello che è difficile da realizzare può essere inserito nella cache.

## 12.4.5 Suggerimenti

Le entrate della cache sono sempre corrette: una ricerca sulla cache può fallire, ma se si trova una entrata, quella entrata è sicuramente corretta, e può essere utilizzata senza ulteriori difficoltà. In alcuni sistemi, è conveniente avere una tabella di suggerimenti: questi sono consigli per la soluzione, ma non è garantito che siano corretti, quindi il chiamante deve verificare da sé il risultato.

Un esempio ben noto di suggerimenti sono gli URL collegati alle pagine Web. Cliccare su un collegamento non garantisce che la pagina Web a cui il collegamento punta sia disponibile; infatti la pagina a cui punta potrebbe essere stata rimossa anni prima, quindi l'informazione sulla pagina puntata è realmente solo un suggerimento.

I suggerimenti sono anche usati per collegarsi ai file remoti: il suggerimento dice qualcosa sul file remoto, ad esempio dove era localizzato, tuttavia, il file può essere stato spostato o cancellato dopo aver registrato il suggerimento, quindi è sempre necessario un controllo per vedere se il suggerimento è corretto.

## 12.4.6 Sfruttare la località

I processi ed i programmi non agiscono a caso, presentano una notevole quantità di località nel tempo e nello spazio, e quest'informazione può essere sfruttata in vari modi per migliorare le prestazioni. Un esempio ben noto di località spaziale è il fatto che i processi non saltano a caso all'interno dei loro spazi di indirizzamento, ma tendono ad usare un numero relativamente piccolo di pagine durante un dato intervallo di tempo. Le pagine che un processo sta utilizzando attivamente possono essere viste come il suo insieme di lavoro (working set), ed il sistema operativo può garantire che quando si manda in esecuzione il processo, il suo working set stia in memoria, riducendo, quindi, il numero dei fault di pagina.

Il principio di località vale anche per i file. Quando un processo ha selezionato una particolare directory di lavoro, è probabile che molti dei suoi riferimenti a file successivi siano proprio ai file di quella directory; infatti, mettendo vicini sul disco tutti gli i-node ed i file di ogni directory, si possono ottenere miglioramenti delle prestazioni. Questo principio è alla base del Berkeley Fast File System (McKusick ed altri, 1984).

Un altro campo in cui la località gioca un ruolo importante è nella schedulazione dei thread nei multiprocessori: come abbiamo visto nel Capitolo 8, un modo per schedulare i thread su un multiprocessore è quello di cercare di far girare ogni thread sulla CPU che ha utilizzato per ultima, nella speranza che alcuni dei suoi blocchi di memoria siano già nella cache di memoria.

## 12.4.7 Ottimizzare il caso comune

È spesso una buon'idea distinguere tra il caso più comune ed il caso peggiore possibile, e trattarli in maniera differente, poiché spesso nei due casi il codice è piuttosto differente; inoltre, è importante rendere veloce il caso più comune, e per quanto riguarda il caso peggiore, se si verifica raramente, è sufficiente gestirlo correttamente.

Come primo esempio, si consideri l'ingresso in una regione critica: la maggior parte delle volte l'ingresso ha successo, specialmente se i processi non passano troppo tempo all'interno delle regioni critiche. Windows 2000 trae dei vantaggi da questa ipotesi, fornendo una chiamata API di Win 32 `EnterCriticalSection` che controlla un flag in modo utente (usando TSL o equivalente). Se il test ha successo, il processo entra semplicemente nella regione critica e non è necessaria nessun chiamata del nucleo; se il test fallisce, la procedura di libreria effettua una down su un semaforo per bloccare il processo. Quindi, nel caso normale non è necessaria nessuna chiamata del nucleo.

Come secondo esempio, si consideri l'impostazione di un allarme (utilizzando segnali in UNIX): se nessun allarme è pendente, è facile inserire una entrata nella coda dei timer, ma se c'è un allarme pendente, bisogna trovarlo e rimuoverlo dalla coda dei timer. Poiché la chiamata `alarm` non specifica se esiste un allarme già impostato, il sistema deve ipotizzare il caso peggiore, cioè che ci sia; tuttavia, poiché la maggior parte delle volte non vi sono allarmi pendenti, e poiché rimuovere un allarme è costoso, è una buona idea distinguere questi due casi.

Un modo per effettuare questo è di tenere un bit nella tabella dei processi che indica se un allarme è pendente o meno: se il bit è a zero (off), viene seguito il percorso più semplice (aggiungendo soltanto una nuova entrata nella coda dei timer senza effettuare controlli), se il bit è a uno (on), deve essere controllata la coda dei timer.

## 12.5 Gestione del progetto

I programmati sono eterni ottimisti; la maggior parte di loro pensa che il modo per scrivere un programma è correre alla tastiera ed iniziare a scrivere. In poco tempo, quindi, tutto il programma sarà terminato e verificato. Tuttavia, per programmi molto grandi, le cose non vanno così; nelle sezioni seguenti vogliamo dire qualcosa sulla gestione dei grandi progetti software, specialmente sui grandi progetti dei sistemi operativi.

### 12.5.1 The Mythical Man Month

Fred Brooks, uno dei progettisti del sistema operativo OS/360 che successivamente passò all'università, nel suo libro classico (Brooks, 1975, 1995) affrontò il problema del perché è così difficile costruire grandi sistemi operativi. Quando la maggior parte dei programmati legge la sua affermazione che i programmati possono produrre, nei grandi progetti, solo 1000 linee di codice senza bachi all'anno, si domanda se il Prof. Brooks vive in uno spazio lontano, magari sul Pianeta Baco; dopo tutto, la maggior parte di loro può ricordare un nottambulo che ha prodotto un programma di 1000 linee in una notte. Come può la stessa cifra essere l'output annuale di una persona con quoziente d'intelligenza sopra 50?

Quello che Brooks sottolineava era che i progetti grandi, con centinaia di programmati, sono completamente differenti dai progetti piccoli, e che quindi i risultati ottenuti

dai progetti piccoli non sono paragonabili a quelli ottenuti dai progetti grandi. In un progetto grande, viene speso un enorme ammontare di tempo per pianificare come dividere il lavoro in moduli, specificando attentamente i moduli e le loro interfacce, e cercando di immaginare, prima di iniziare il codice, come interagiranno i vari moduli; quindi, si devono codificare i moduli, e si deve effettuare su di loro il debug indipendentemente dagli altri; infine, i moduli devono essere integrati ed il sistema deve essere testato nel complesso. Il caso normale è che ogni modulo, quando viene testato singolarmente, lavori perfettamente, ma il sistema fallisce istantaneamente quando tutti i pezzi sono messi insieme. Brooks stimò il lavoro come segue

- 1/3 Progettazione
- 1/6 Codifica
- 1/4 Test dei moduli
- 1/4 Test del sistema

In altre parole, la scrittura del codice è la parte più facile, mentre la parte più difficile consiste nel comprendere cosa dovrebbero fare i moduli, e fare in modo che il modulo A comunichi correttamente con il modulo B; in un programma piccolo scritto da un singolo programmatore, c'è da fare solo la parte semplice.

Il titolo del libro di Brooks deriva dalla sua affermazione che le persone ed il tempo non sono intercambiabili: non esiste l'unità di lavoro mese-uomo (o mese-persona). Se per costruire un progetto si impiegano 15 persone per 2 anni, è inconcepibile che 360 persone possano effettuarlo in un mese e probabilmente non è possibile avere 60 persone che lo facciano in 6 mesi.

Ci sono tre motivi per spiegare quest'effetto: primo, il lavoro non può essere interamente parallelizzato; fino a quando la progettazione non è completa, e finché non sono stati determinati i moduli necessari e come devono essere le loro interfacce, non si può iniziare a scrivere nessun codice. In un progetto di due anni, la sola progettazione può richiedere 8 mesi.

Secondo, per utilizzare interamente un grande numero di programmatore, il lavoro deve essere partizionato in un grande numero di moduli, in modo che ognuno abbia qualcosa da fare. Poiché ogni modulo può potenzialmente interagire con ogni altro modulo, il numero di interazioni modulo-modulo che è necessario considerare, cresce come il quadrato del numero dei moduli, cioè, come il quadrato del numero dei programmatore. Questa complessità può facilmente sfuggire di mano; infatti, alcune misure accurate eseguite su 63 progetti software hanno confermato che, su grandi progetti, il compromesso tra le persone ed i mesi è lontano dall'essere lineare (Boehm, 1981).

Terzo, il controllo dei possibili bachi è fortemente sequenziale. Mettere 10 persone a controllare gli errori in un problema, non consente di scovare l'errore 10 volte più velocemente; infatti, 10 persone che controllano gli errori sono probabilmente più lente di uno solo, in quanto questi sprecheranno più tempo a comunicare l'uno con l'altro.

Brooks ha riassunto la sua esperienza sul compromesso tra le persone ed il tempo nella legge di Brooks:

*Aggiungere forza lavoro ad un progetto software in ritardo lo rende ancora più in ritardo.*

Il problema dato dall'aggiungere persone è che questi devono prepararsi al progetto, i moduli devono essere nuovamente divisi per poter essere destinati ad un numero più grande di programmatore disponibili ed inoltre saranno necessari molti incontri per coordinare tutti gli sforzi, e così via; Abdel-Hamid e Madnick (1991) hanno confermato

questa legge in maniera sperimentale. Un modo leggermente irriverente per esprimere in altra forma la legge di Brooks è

*Servono 9 mesi per fare un bambino, qualunque sia il numero delle donne che si impiegano in questo lavoro.*

## 12.5.2 Struttura del gruppo di lavoro

I sistemi operativi commerciali sono grandi progetti software e richiedono inevitabilmente grandi gruppi di persone; e la qualità delle persone incide immensamente. È noto da decenni che i programmatore migliori sono 10 volte più produttivi dei programmatore scarsi (Sackman ed altri, 1968), il problema però è che, quando servono 200 programmatore, è difficile trovare 200 programmatore validi; ci si dovrà preparare ad un vasto spettro di qualità.

Un'altra cosa importante, in un ogni grande progetto, software o meno, è la necessità di una coerenza architettonale: ci dovrebbe essere una sola mente che controlla il progetto. Brooks cita la cattedrale di Reims come esempio di grande progetto che è stato costruito in decine di anni, ed in cui gli architetti che vennero dopo misero da parte il loro desiderio di dare una impronta personale al progetto, per portare a termine i piani dell'architetto precedente: il risultato è una coerenza architettonale che non trova paragoni nelle altre cattedrali europee.

Negli anni 70, Harlan Mills mise insieme l'osservazione che alcuni programmatore sono migliori di altri, con la necessità di coerenza architettonale, proponendo il paradigma del **gruppo di lavoro con programmatore capo** (chief programmer team, Baker, 1972): la sua idea era quella di organizzare un gruppo di programmazione come una equipe chirurgica, invece che come un gruppo di addetti alla macellazione del maiale. Invece di tagliuzzare tutti come dei pazzi, una sola persona tiene il bisturi, e tutti gli altri gli devono fornire supporto. Per un progetto di 10 persone, Mills suggeriva una struttura di gruppo come quella di Figura 12.9.

Sono passati tre decenni dal momento in cui queste cose sono state proposte ed inserite nella produzione, ed alcune cose sono cambiate (come ad esempio la necessità di avere un avvocato del linguaggio - C è più semplice di PL/I), ma la necessità di avere una sola mente che controlla il progetto è ancora vera, e anche il fatto che tale mente dovreb-

| Titolo                  | Compito                                                                                   |
|-------------------------|-------------------------------------------------------------------------------------------|
| Programmatore capo      | Esegue il progetto architettonale e scrive il codice                                      |
| Co-pilota               | Aiuta il programmatore capo e serve per scambiare opinioni                                |
| Amministratore          | Gestisce le persone, il bilancio, lo spazio, l'equipaggiamento, i rapporti, eccetera      |
| Editore                 | Prepara la documentazione, che deve essere scritta dal programmatore capo                 |
| Segretarie              | L'amministratore e l'editore necessitano di una segretaria ciascuno                       |
| Impiegato di programma  | Mantiene gli archivi del codice e della documentazione                                    |
| Fabbro degli strumenti  | Fornisce ogni strumento che serve al programmatore capo                                   |
| Tester                  | Testa il codice del programmatore capo                                                    |
| Avvocato del linguaggio | Lavoratore part-time che può consigliare il programmatore capo a proposito del linguaggio |

Figura 12.9 Proposta di Mills per formare un gruppo di lavoro con 10 programmatore e un capo.

essere in grado di lavorare al 100% per la progettazione e per la programmazione, da cui segue la necessità di avere uno staff di supporto; sebbene ora, con l'aiuto del computer, sia sufficiente uno staff più piccolo, essenzialmente l'idea è ancora valida.

Ogni grande progetto deve essere organizzato gerarchicamente: al livello più basso ci sono molti gruppi di lavoro di piccole dimensioni, ognuno con a capo un programmatore capo; al livello successivo, gli insiemi di gruppi di lavoro devono essere coordinati da un direttore. L'esperienza mostra che dirigere una persona costa il 10% del tempo del direttore, quindi è necessario un direttore a tempo pieno per ogni insieme di 10 gruppi di lavoro; questi direttori devono essere a loro volta diretti, e così via salendo nell'albero.

Brooks osservò che le cattive notizie non salgono bene nell'albero della gerarchia. Jerry Saltzer del M.I.T. chiamò questo effetto il **diodo delle cattive notizie**: nessun programmatore capo o direttore vuole dire al suo rispettivo superiore che il progetto è in ritardo di 4 mesi, e che non ha nessuna possibilità di rispettare la scadenza, perché c'è la tradizione, da 1000 anni, di decapitare i messaggeri che forniscono cattive notizie. Di conseguenza, il direttore al vertice della gerarchia è generalmente all'oscuro dello stato del progetto; e quando diventa ovvio che la scadenza non può essere rispettata, i direttori al vertice tentano di porre rimedio aggiungendo persone, e a quel punto interviene la legge di Brooks.

In pratica, le grandi aziende, che hanno una lunga esperienza nella produzione del software e che conoscono cosa accade se è prodotto a casaccio, tendono a cercare almeno di farlo bene. Al contrario, le nuove, piccole aziende, che hanno un'enorme fretta di accaparrarsi il mercato, non hanno sempre l'accortezza di produrre il loro software attentamente, e questa fretta spesso porta ad ottenere risultati ben lontani dall'ottimo.

Né Brooks né Mills hanno previsto la crescita del movimento per il codice aperto al pubblico. Sebbene abbia avuto qualche successo, resta da verificare se questo modello è valido per produrre grandi software di qualità, una volta che le novità sono svariate. Si ricordi che in passato, la radio era dominata da operatori radioamatori, ma presto prese campo la radio commerciale, e successivamente la televisione commerciale. Quello che si può notare è che i progetti software aperti al pubblico che hanno avuto molto successo, hanno chiaramente utilizzato il modello del programmatore capo per avere una sola mente che controlla il progetto architettonico (per esempio Linus Torvalds per il kernel Linux e Richard Stallman per il compilatore C della GNU).

### 12.5.3 Il ruolo dell'esperienza

Avere progettisti esperti è un elemento critico per un progetto di un sistema operativo; Brooks fa notare che la maggior parte degli errori non è nel codice, ma nel progetto: i programmatore eseguono correttamente quello che gli viene detto di fare, ma se quello che gli viene detto di fare è sbagliato, nessun tipo di test software può catturare le specifiche sbagliate.

La soluzione di Brooks è di abbandonare il modello di sviluppo classico di Figura 12.10(a) e di utilizzare il modello di Figura 12.10(b): l'idea è di cercare prima di tutto di scrivere un programma principale che chiama semplicemente le procedure di alto livello, che sono inizialmente fittizie. A partire dal primo giorno di progettazione, il sistema verrà compilato e verrà eseguito, sebbene non faccia nulla; con il passare del tempo, si inseriscono i moduli nel sistema intero; il risultato di questo approccio è che i test per l'integrazione del sistema vengono eseguiti continuamente, quindi gli errori nel progetto vengono fuori molto presto. In effetti, il processo di riconoscimento delle decisioni di progetto errate inizia molto presto nel ciclo.

Conoscere poco è pericoloso. Brooks aveva osservato quello che lui chiamava il **l'ef-**

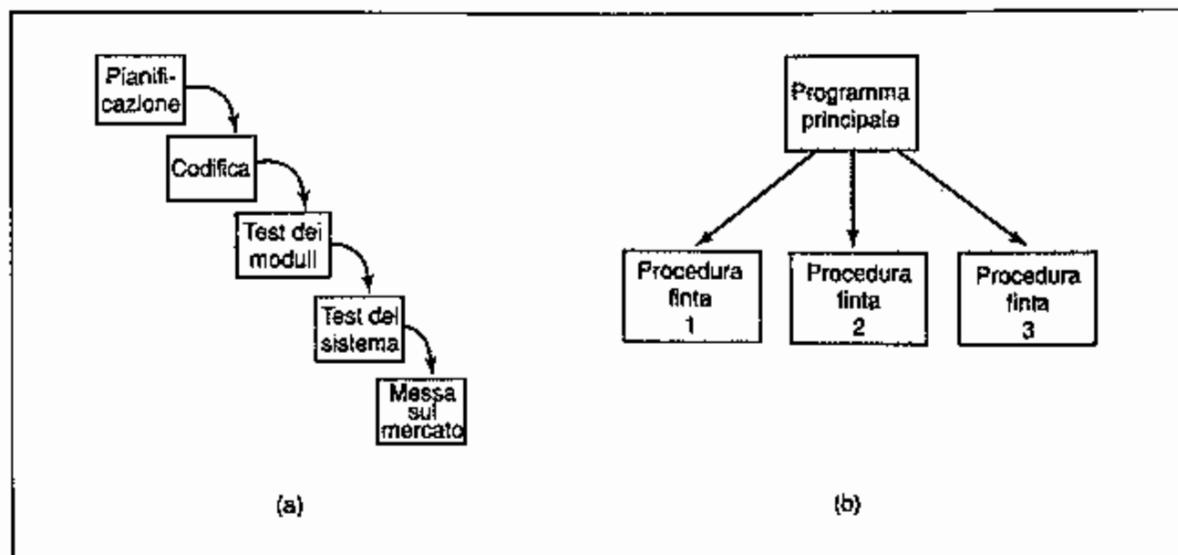


Figura 12.10 (a) Un progetto software tradizionale che progredisce a stadi. (b) Un progetto alternativo che produce un sistema eseguibile (che non fa nulla) iniziando dal primo giorno.

**fatto del secondo sistema:** spesso il primo prodotto realizzato da un gruppo di progetto è minimale, poiché i progettisti erano preoccupati che potesse non funzionare affatto; di conseguenza, esitavano ad inserire nel progetto molte caratteristiche. Quando un progetto ha successo, i progettisti costruiscono il sistema successivo, e incoraggiati dai loro successi, la seconda volta includono tutti i fronzoli che avevano di proposito evitato di inserire la prima volta: come risultato, il secondo sistema diventa eccessivamente grosso e ha prestazioni scadenti. La terza volta i progettisti sono rinsaviti dopo il fallimento del secondo sistema, e sono nuovamente cauti.

La coppia CTSS-MULTICS è un chiaro esempio di quanto detto sopra: CTSS fu il primo sistema general-purpose a condivisione di tempo, e fu un enorme successo pur avendo funzionalità minimali; il suo successore, MULTICS era troppo ambizioso e soffrì molto per questo; infatti, le idee erano valide, ma c'erano troppe cose nuove, quindi il sistema funzionò per anni con prestazioni scarse e non fu mai un grande successo commerciale. Il terzo sistema in questa linea di sviluppo, UNIX, fu molto più cauto ed ebbe molto più successo.

#### 12.5.4 Nessuna pallottola è d'argento

In aggiunta all'articolo *The Mythical Man Month*, Brooks scrisse un autorevole articolo chiamato "No Silver Bullet" (Nessuna pallottola è d'argento) (Brooks, 1987). In esso, affermò che nessuno dei molti toccasana sperimentati in un decennio da diverse persone all'epoca, erano serviti a generare un miglioramento di un ordine di grandezza nella produttività del software: l'esperienza di nostra che aveva ragione. Fra le pallottole d'argento che furono proposte, c'erano i linguaggi di programmazione ad alto livello, la programmazione orientata agli oggetti, l'intelligenza artificiale, i sistemi esperti, la programmazione automatica, la programmazione grafica, la verifica automatica del programma, e gli ambienti di programmazione. Forse il prossimo decennio vedrà una pallottola d'argento, ma forse dovremo realizzare miglioramenti graduuali ed incrementali.

## 12.6 Tendenze nella progettazione di un sistema operativo

Fare previsioni è sempre difficile – specialmente riguardo al futuro. Per esempio, nel 1899, il capo dell'Ufficio Brevetti statunitense, Charles H. Duell, chiese all'allora Presidente McKinley di abolire l'Ufficio Brevetti (ed il suo lavoro!), poiché egli disse: "Ogni cosa che può essere inventata, è già stata inventata" (Cerf e Navasky, 1984); nonostante questo, Thomas Edison, pochi anni dopo, bussò alla sua porta con un paio di nuovi brevetti, tra cui la luce elettrica, il fonografo e il proiettore di pellicole. Proviamo a lucidare la nostra sfera di cristallo, ed avventuriamoci ad indovinare cosa ci riserveranno i sistemi operativi nel prossimo futuro.

### 12.6.1 Sistemi operativi con un grande spazio di indirizzamento

Poiché le macchine sono passate da uno spazio di indirizzamento a 32 bit ad uno spazio di indirizzamento a 64 bit, diventano possibili grandi cambiamenti nella progettazione di un sistema operativo. Uno spazio di indirizzamento a 32 bit non è poi molto grande, infatti se si cerca di dividere  $2^{32}$  byte dando ad ogni persona della terra il suo byte, potrebbero non esserci abbastanza byte disponibili; al contrario,  $2^{64}$  è circa  $2 \times 10^{19}$  e così facendo tutti avranno la loro personale grossa fetta di 3GB.

Cosa potremmo fare con uno spazio di indirizzamento di  $2 \times 10^{19}$  byte? Per iniziare, potremmo eliminare il concetto di file system, infatti, tutti i file potrebbero essere mantenuti concettualmente in memoria (virtuale) per tutto il tempo; dopo tutto, c'è abbastanza spazio per più di un miliardo di intere pellicole, ognuna compressa a 4GB.

Un altro possibile utilizzo è quello di avere un deposito di oggetti persistenti. Gli oggetti potrebbero essere creati nello spazio di indirizzamento, ed esservi mantenuti fino a quando non vi sono più riferimenti ad essi; a quel punto potrebbero essere automaticamente cancellati. Tali oggetti potrebbero essere persistenti nello spazio di indirizzamento, anche durante operazioni di shutdown e reboot del computer, inoltre con uno spazio di indirizzamento a 64 bit, gli oggetti potrebbero essere creati al tasso di 100 MB/s per 5000 anni, prima di superare lo spazio di indirizzamento a disposizione. Di sicuro, per immagazzinare davvero questa quantità di dati, sarebbe necessario un grande spazio su disco per il traffico di paginazione, ma per la prima volta nella storia, il fattore di limitazione sarebbe lo spazio su disco e non lo spazio d'indirizzamento.

Con un gran numero di oggetti nello spazio di indirizzamento, diventa interessante consentire a diversi processi di girare nello stesso spazio di indirizzamento allo stesso momento, per condividere gli oggetti in modo generale. Tale progetto porta chiaramente ad avere sistemi operativi veramente differenti da quelli che abbiamo ora; alcune riflessioni su questo concetto sono contenute in Chase ed altri, 1994.

Un altro concetto sui sistemi operativi che avremo da rivedere con indirizzi a 64 bit è la memoria virtuale: con  $2^{64}$  byte di spazio di indirizzamento virtuale, e pagine da 8 KB, abbiamo  $2^{51}$  pagine. Le tabelle delle pagine convenzionali non si adattano bene a queste dimensioni, quindi è necessario qualcos'altro; ad esempio, una possibilità sono le tabelle delle pagine invertite, ma sono state proposte anche altre idee (Talluri ed altri, 1995); in ogni caso, c'è uno spazio enorme per nuove ricerche su sistemi operativi a 64 bit.

## 12.6.2 Reti

I sistemi operativi attuali furono progettati per computer indipendenti; le reti sono state un'idea successiva e generalmente si accede ad esse attraverso programmi speciali, come ad esempio browser Web, FTP, o telnet; ma nel futuro, le reti saranno probabilmente alla base di tutti i sistemi operativi. Un computer indipendente senza una connessione di rete sarà raro come un telefono senza una connessione alla rete telefonica, e probabilmente sarà nella norma avere connessioni a molti megabit al secondo.

I sistemi operativi dovranno cambiare per adattarsi a questo cambiamento di paradigma. La differenza tra dati locali e dati remoti può sfumare, fino al punto in cui virtualmente nessuno conosce o si interessa a dove siano memorizzati i dati: qualsiasi computer, ovunque esso sia, è in grado di gestire i dati sparsi per la rete come dati locali. In modo limitato, questo è già vero con NFS, ma probabilmente questo concetto diventerà molto più diffuso e molto meglio integrato.

L'accesso al Web, che ora richiede programmi speciali (browser) può diventare anch'esso completamente integrato nel sistema operativo, in modo continuo. Le pagine Web possono diventare il modo standard di memorizzare delle informazioni, e queste pagine possono contenere una grande varietà di elementi non testuali, tra cui audio, video, programmi, e molto altro, tutti gestiti come dati fondamentali del sistema operativo.

## 12.6.3 Sistemi paralleli e distribuiti

Un'altra area attiva che sta prendendo campo riguarda i sistemi paralleli e distribuiti. I sistemi operativi attuali per multiprocessori e multicompiler sono soltanto dei sistemi operativi standard monoprocessoressi con minori variazioni allo schedulatore per gestire un po' meglio il parallelismo. Nel futuro, potremo vedere sistemi operativi in cui il parallelismo è molto più centrale di come è adesso, e questo effetto sarà estremamente stimolato se le macchine da tavolo avranno presto due, quattro o più CPU in una configurazione a multiprocessore. Questo può consentire di progettare molti programmi applicativi per multiprocessori, con l'esigenza concomitante di avere, per questi, un migliore supporto dal sistema operativo.

I multicompiler probabilmente prevorranno nei prossimi anni sui supercomputer che si usano per calcoli scientifici ed ingegneristici su vasta scala; ma i loro sistemi operativi sono ancora troppo primitivi: la distribuzione dei processi, il bilanciamento del carico, e le comunicazioni necessitano ancora di un grande lavoro.

I sistemi distribuiti attuali sono spesso costruiti come middleware, poiché i sistemi operativi esistenti non forniscono facilitazioni adatte alle applicazioni distribuite, mentre in futuro i sistemi operativi potranno essere progettati tenendo a mente i sistemi distribuiti, in modo da includere fin dall'inizio le caratteristiche necessarie.

## 12.6.4 Sistemi multimediali

I sistemi multimediali sono chiaramente una stella emergente nel mondo dei computer. Non dovrebbe sorprendere nessuno che i computer, gli stereo, le televisioni, ed i telefoni vengano uniti insieme in un singolo dispositivo, capace di supportare anche immagini di alta qualità, audio e video, e vengano connessi a reti ad alta velocità, in modo che questi file possano facilmente essere scaricati, scambiati ed acceduti in maniera remota. I sistemi operativi per questi dispositivi, o anche per dispositivi audio e video indipendenti, dovranno essere sostanzialmente differenti dai sistemi attuali; in particolare, saranno necessarie delle garanzie di tempo reale, e queste guideranno la progettazione.

ne del sistema. Inoltre, i consumatori non tolleranno affatto guasti, che durino qualche ora, ai loro televisori digitali, quindi saranno indispensabili software di maggiore qualità e meccanismi di tolleranza ai guasti; siccome poi i file multimediali tendono ad essere molto grandi, i file system dovranno cambiare per essere in grado di gestire questi file in maniera efficiente.

## 12.6.5 Computer alimentati a batteria

Tra non molto i PC da tavolo potenti, probabilmente con spazi di indirizzamento a 64 bit, connessione in rete a grande larghezza di banda, processori multipli, e audio e video di alta qualità, diventeranno molto comuni, senza dubbio, ed i loro sistemi operativi dovranno essere significativamente diversi dai sistemi attuali, per poter gestire queste richieste. Tuttavia, una porzione in crescita ancora più rapida sul mercato, riguarda i computer alimentati a batteria, compresi i computer portatili e i computer palmari, le tavolette Web e i vari telefoni ibridi. Alcuni di questi avranno connessioni con il mondo esterno senza fili; altri gireranno in modo non connesso quando non saranno collegati in casa. Tutti avranno bisogno di sistemi operativi differenti, che siano più piccoli, più veloci, più flessibili e più affidabili dei sistemi attuali; e qui possono trovare campo i vari tipi di microkernel e i sistemi estensibili.

Questi sistemi operativi dovranno gestire meglio dei sistemi odierni le operazioni in modalità completamente connessa (per esempio con fili), o debolemente connessa (per esempio senza fili) e senza connessione, il che comprende ottenere i dati prima di scollegarsi, ed analizzare la consistenza quando si torna ad essere collegati. Inoltre essi devono anche gestire i problemi di mobilità in maniera migliore dei sistemi attuali (per esempio, trovare una stampante laser, registrarsi su questa, e spedirgli un file attraverso la radio). La gestione dell'alimentazione sarà essenziale, e saranno fondamentali le comunicazioni tra sistema operativo ed applicazioni per gestire il consumo della batteria, ed utilizzarla al meglio. Possono diventare importanti anche adattamenti dinamici delle applicazioni per gestire limitazioni dovuti agli schermi piccoli, infine, le nuove modalità di input e output, tra cui il riconoscimento della scrittura manuale e del linguaggio parlato, possono richiedere nuove tecniche nel sistema operativo per migliorarne la qualità. È improbabile che un sistema operativo per un computer alimentato a batteria, tenuto in mano, senza fili, con comandi vocali abbia molto in comune con quello di un computer da tavolo multiprocessore a quattro CPU a 64 bit con una connessione di rete a fibre ottiche di un gigabit; e sicuramente, ci saranno innumerevoli macchine ibride con le loro richieste particolari.

## 12.6.6 Sistemi embedded

Un ultimo campo in cui i nuovi sistemi operativi prolifereranno riguardano i sistemi embedded. I sistemi operativi inseriti nelle lavatrici, nei forni a microonde, nelle bambole, nelle radio a transistor (Internet?), nei lettori MP3, nelle telecamere, negli ascensori e nei pacemaker saranno diversi da tutti i precedenti e probabilmente anche tra di loro; ognuno di essi sarà probabilmente adattato in maniera scrupolosa alla sua applicazione specifica, poiché non è probabile che qualcuno inserisca una scheda PCI in un pacemaker per farlo diventare un controllore di ascensore. Poiché tutti i sistemi embedded eseguono soltanto un numero limitato di programmi, conosciuti a tempo di progettazione, può essere possibile effettuare delle ottimizzazioni non consentite nei sistemi general-purpose.

Una idea promettente per i sistemi embedded sono i sistemi operativi estensibili (per esempio Paramecium e Exokernel); questi possono diventare leggeri o pesanti, a seconda

di cosa richiede l'applicazione in questione, ma in modo coerente con le applicazioni. Visto che i sistemi embedded saranno prodotti a centinaia di milioni, per i sistemi operativi questo sarà un nuovo grande mercato.

## 12.7 Sommario

La progettazione di un sistema operativo inizia con il determinare cosa questo dovrebbe fare. L'interfaccia dovrebbe essere semplice, completa ed efficiente; si dovrebbe avere un paradigma chiaro di interfaccia utente, un paradigma di esecuzione ed un paradigma dei dati.

Il sistema dovrebbe essere ben strutturato, utilizzando una delle diverse tecniche conosciute, come ad esempio la struttura a strati o la modalità client-server; inoltre i componenti interni dovrebbero essere ortogonali gli uni agli altri, ed avere chiaramente separate le politiche dai meccanismi. Si dovrebbero studiare attentamente alcuni problemi, come ad esempio le strutture dati statiche oppure quelle dinamiche, la gestione dei nomi, il tempo di binding, e l'ordine di implementazione dei moduli.

Le prestazioni sono importanti, ma le ottimizzazioni dovrebbero essere scelte attentamente in modo da non rovinare la struttura del sistema; inoltre vale la pena di fare compromessi tra spazio e tempo, utilizzare una cache o i suggerimenti, sfruttare la località e ottimizzare il caso più comune.

Scrivere un sistema con un paio di persone è diverso dal produrre un grande sistema con 300 persone: in quest'ultimo caso, la struttura del gruppo di lavoro e la gestione del progetto giocano un ruolo cruciale nel successo o nel fallimento del progetto stesso.

Infine, i sistemi operativi cambieranno negli anni a venire, per seguire le nuove tendenze ed affrontare nuove sfide: queste comprenderanno gli spazi di indirizzamento a 64 bit, la connettività a larga banda, i multiprocessori da tavolo, la multimedialità, i computer palmari senza fili, e una enorme varietà di sistemi embedded. Gli anni a venire saranno quindi anni eccitanti per i progettisti di sistemi operativi.

## PROBLEMI

1. La legge di Moore descrive un fenomeno di crescita esponenziale, simile alla crescita della popolazione di una specie animale, introdotta in un nuovo ambiente con cibo abbondante e nessun nemico naturale. In natura, una curva di crescita esponenziale probabilmente prima o poi diventa una curva sigmoidea, con un limite asintotico quando il rifornimento di cibo diventa limitato, o i predatori imparano ad acciuffare le nuove prede. Discutere alcuni fattori che possono prima o poi limitare il tasso di miglioramento dell'hardware di un computer.
2. In Figura 12.1, sono mostrati due paradigmi, algoritmico e orientato agli eventi. Per ognuno dei seguenti tipi di programma, quale paradigma sembra essere più facile da utilizzare?
  - (a) Un compilatore.
  - (b) Un programma di editing di fotografie.
  - (c) Un programma per il pagamento delle buste paga.
3. In alcuni dei primi Apple Macintosh, il codice della GUI era nella ROM. Perché?

4. I nomi gerarchici dei file iniziano sempre dal vertice dell'albero: si consideri, ad esempio, il nome del file `/usr/ast/libro/mos2/cap-12` invece di `/cap-12/mos2/libro/ast/usr`. Al contrario, i nomi del DNS iniziano dal fondo dell'albero e vanno verso l'alto. C'è una ragione fondamentale per questa differenza?
5. Il principio di Corbató afferma che il sistema dovrebbe fornire un meccanismo minimale. Di seguito abbiamo una lista di chiamate POSIX che erano anche presenti nella Versione 7 di UNIX; quale di queste è ridondante, cioè, potrebbe essere rimossa senza nessuna perdita di funzionalità, poiché combinazioni semplici delle altre chiamate potrebbero effettuare lo stesso lavoro, con quasi le stesse prestazioni? `Access`, `alarm`, `chdir`, `chmod`, `chown`, `chroot`, `close`, `creat`, `dup`, `exec`, `exit`, `fcntl`, `fork`, `fstat`, `ioctl`, `kill`, `link`, `lseek`, `mkdir`, `open`, `pause`, `pipe`, `read`, `stat`, `times`, `umask`, `unlink`, `utime`, `wait` e `write`.
6. Si supponga che gli strati 3 e 4 di Figura 12.2 vengano scambiati. Quali implicazioni avrebbe tale scambio nella progettazione del sistema?
7. In un sistema client-server basato su microkernel, il microkernel esegue soltanto lo scambio di messaggi e niente altro. È possibile che i processi utente, ciononostante, creino ed utilizzino i semafori? Se sì, come? Se no, perché?
8. Le prestazioni delle chiamate di sistema possono migliorare con accurate ottimizzazioni. Si consideri il caso in cui una chiamata di sistema sia effettuata ogni 10 ms; sapendo che il tempo medio di una chiamata è 2 ms, se le chiamate di sistema sono velocizzate di un fattore due, quanto tempo impiegherà un processo di 10 s, per essere eseguito?
9. Dare una breve discussione del concetto di "meccanismi contro politiche" nel contesto dei magazzini di vendita al dettaglio.
10. I sistemi operativi spesso effettuano l'attribuzione dei nomi a due diversi livelli: esterno ed interno. Quali sono le differenze tra questi nomi rispetto ai seguenti elementi:
  - (a) Lunghezza
  - (b) Unicità
  - (c) Gerarchia
11. Un modo per gestire le tabelle la cui dimensione non è conosciuta in anticipo è quella di renderle fisse, ma quando una si riempie, la si rimpiazza con una più grande, copiando le vecchie entrate sulle nuove, e quindi si rilascia la vecchia tabella. Quali sono i vantaggi e gli svantaggi, se si costruisce la nuova tabella grande 2 volte la dimensione della tabella originale, piuttosto che se la si costruisce grande una volta e mezza la tabella originale?
12. In Figura 12.4 viene utilizzata una variabile, `trovato`, per indicare se il PID è allocato. Sarebbe possibile dimenticare `trovato` e limitarsi a controllare soltanto `p` alla fine del ciclo per vedere se è arrivato alla fine o no?
13. In Figura 12.5, le differenze tra il Pentium e l'UltraSPARC vengono nascoste dalla compilazione condizionale. Si può utilizzare lo stesso approccio per nascondere la differenza tra i Pentium con un disco IDE come unico disco, e i Pentium con un disco SCSI come unico disco? Sarebbe una buona idea?
14. L'indirezione è un modo per rendere un algoritmo più flessibile. Questo porta a degli svantaggi? Se sì, quali?

15. Le procedure rientranti possono avere variabili globali private statiche? Discuti la risposta.
16. La macro di Figura 12.6(b) è chiaramente molto più efficiente della procedura di Figura 12.6(a). Uno svantaggio, tuttavia, è il fatto che la macro è più difficile da leggere. Ci sono altri svantaggi? Se sì, quali sono?
17. Si supponga di avere bisogno di un modo per calcolare se il numero di bit in una parola di 32 bit è pari o dispari; trovare un algoritmo per eseguire questo calcolo il più velocemente possibile (si possono usare 256 KB di RAM per le tabelle che servono). Scrivere una macro per realizzare tale algoritmo. *Facoltativo:* Scrivere una procedura che esegua il calcolo con un ciclo sui 32 bit, e misurare di quale fattore è più veloce la macro rispetto alla procedura.
18. In Figura 12.7 abbiamo mostrato come i file GIF utilizzano valori di 8-bit per indicizzare una tavolozza di colori; la stessa idea può essere utilizzata con una tavolozza di colori di 16 bit. In quali ipotesi, se ce ne sono, potrebbe essere una buona idea avere una tavolozza di colori di 24 bit?
19. Uno svantaggio delle immagini GIF è che l'immagine deve includere la tavolozza dei colori, il che aumenta la grandezza del file. Qual è la dimensione minima dell'immagine per cui una tavolozza di colori da 8 bit non dà nessun vantaggio? Rispondere alla domanda anche per una tavolozza di colori da 16 bit.
20. Nel testo abbiamo mostrato come inserire nella cache i nomi di percorso, e come questo meccanismo possa realizzare un significativo aumento della velocità quando si ricercano i nomi di percorso. Un'altra tecnica che è utilizzata qualche volta è quella di avere un programma demone che apre tutti i file nella directory principale, e li tiene aperti in permanenza, per forzare che i loro i-node stiano in memoria per tutto il tempo. Questa operazione sugli i-node può migliorare ulteriormente la ricerca di percorso?
21. Anche se un file remoto non è stato rimosso dopo aver registrato il suggerimento, potrebbe essere cambiato dall'ultimo riferimento. Quali altre informazioni potrebbe essere utile registrare?
22. Si consideri un sistema che accumula i riferimenti a file remoti come suggerimenti, per esempio come terne (nome, host remoto, nome remoto). È possibile che un file remoto possa essere rimosso senza che l'informazione sia divulgata, e quindi sia sostituito; il suggerimento può, quindi, restituire il file errato. Come si può rendere meno probabile che capiti questo problema?
23. Nel testo è stato detto che la località può spesso essere sfruttata per migliorare le prestazioni, ma si consideri il caso in cui un programma legge dell'input da una sorgente e manda continuamente dell'output a due o più file. In questo caso, un tentativo di sfruttare il vantaggio della località nel file system, può portare ad una diminuzione dell'efficienza? C'è un modo per evitare questo?
24. Fred Brooks sostiene che un programmatore può scrivere annualmente 1000 linee di codice corrette, ciononostante la prima versione di MINIX (13000 linee di codice) fu prodotta da una persona in meno di tre anni. Come spieghi questa discrepanza?
25. Utilizzando il dato di Brooks di avere 1000 linee di codice per programmatore all'anno, stimare il costo dello sviluppo di Windows 2000. Si supponga che un programmatore costi 100000 dollari all'anno (incluse le spese generali, come il compu-

- ter, lo spazio del suo ufficio, il supporto di segretarie e le spese di direzione). Credi a questa risposta? Se no, cosa potrebbe esserci di sbagliato?
- 26. Poiché la memoria diventa sempre meno costosa, si può immaginare di fabbricare un computer con una grande RAM alimentata a batteria invece di avere un hard disk. Ai prezzi attuali, quanto potrebbe costare un PC di fascia bassa con solo una RAM? Si supponga che un disco di 1GB di RAM sia sufficiente per una macchina di fascia bassa. Questa macchina può essere competitiva?
  - 27. Nominare alcune caratteristiche di un sistema operativo convenzionale che non sono necessarie in un sistema embedded all'interno di un elettrodomestico.
  - 28. Scrivere una procedura C per eseguire una addizione a doppia precisione su due parametri dati. Scrivere la procedura utilizzando la compilazione condizionale, in modo che sia eseguibile su macchine a 16 bit e anche su macchine a 32 bit.
  - 29. Scrivere dei programmi che inseriscono in un array stringhe brevi generate a caso, quindi scorrono l'array ricercando una data stringa utilizzando: (a) una ricerca semplice e lineare (forza bruta); (b) un metodo più sofisticato di tua scelta. Ricomporre i programmi partendo da array di piccole dimensioni fino ad array della dimensione massima che può contenere il sistema. Valutare le prestazioni in entrambi gli approcci. Qual è il punto di equivalenza?
  - 30. Scrivere un programma per simulare un file system in memoria.

# LETTURE CONSIGLIATE E BIBLIOGRAFIA

Nei dodici capitoli precedenti si sono toccati vari argomenti; questo capitolo vuole essere un aiuto ai lettori interessati a proseguire ulteriormente lo studio dei sistemi operativi. La Sezione 13.1 è un elenco di letture suggerite; la Sezione 13.2 è una bibliografia in ordine alfabetico di tutti i libri e gli articoli citati in questo libro.

Oltre ai riferimenti dati di seguito, gli *Atti* (in inglese, *Proceedings*, nel seguito abbreviato *Proc.*) dell'*ACM Symposium on Operating Systems Principles* (*SOSP*), organizzato ogni due anni da ACM, e gli *Atti della International Conference on Distributed Computing Systems* (*DCS*), tenuti ogni anno a cura di IEEE, sono buone occasioni per trovare articoli recenti sui sistemi operativi, insieme con l'*USENIX Symposium on Operating Systems Design and Implementation*. Inoltre, *ACM Transactions on Computer Systems* e *Operating Systems Review* sono due riviste che spesso hanno articoli interessanti sui sistemi operativi.

### 13.1 Suggerimenti per ulteriori letture

Nelle sezioni successive, diamo dei suggerimenti per ulteriori letture; a differenza degli articoli citati nella sezioni intitolate "ricerche su ..." nel testo, che trattavano della ricerca attuale, questi riferimenti per la maggior parte risultano introduttivi o tutorial; tuttavia possono servire per presentare il materiale in questo libro da una prospettiva differente o con una diversa enfasi.

### 13.1.1 Introduzione e opere generali

Kavi et al., "Computer Systems Research: The Pressure is On"

In che direzione sta andando la ricerca sui sistemi? Che cos'è importante oggi? Cosa si può dire dello sviluppo di sistemi robusti e prevedibili? E dei chip con miliardi di transistor, o dei sistemi con miliardi di utenti nel mondo? Queste sono alcune delle domande e delle risposte presentate in questo libro.

Milojicic, "Operating Systems: Now and in the Future"

Supponete di poter fare ai sei più grandi esperti nel mondo in sistemi operativi, una serie di domande attinenti a tale campo e a dove stia andando. Otterreste le stesse risposte? Suggerimento: No. Scoprite quello che dicono.

Silberschatz et al., *Operating System Concepts*

Un manuale sui sistemi operativi. Descrive i processi, la gestione della memoria, i sistemi distribuiti e la protezione. Vengono forniti tre casi di studio: UNIX, Linux e Windows NT. La copertina è piena di dinosauri e cosa questo abbia a che fare con i sistemi operativi dell'anno 2000 è poco chiaro.

Stallings, *Operating Systems*. Edizione italiana, *Sistemi operativi*, Jackson Libri 2000.

Un altro manuale sui sistemi operativi: descrive tutti gli argomenti tradizionali, con un po' di materiale sui sistemi distribuiti.

Stevens, *Advanced Programming in the UNIX Environment*

Questo libro spiega come scrivere programmi in C che usino l'interfaccia delle chiamate di sistema UNIX e le librerie standard per C. Gli esempi sono basati su System V Release 4 e la versione di UNIX 4.4BSD. Inoltre viene descritta nel dettaglio la relazione tra queste implementazioni.

### 13.1.2 Processi e thread

Andrews e Schneider, "Concepts and Notations for Concurrent Programming"

Un'introduzione e una rassegna su processi e comunicazione tra processi, tra cui l'attesa attiva (busy waiting), i semafori, i monitor, lo scambio dei messaggi e altre tecniche. L'articolo mostra inoltre come questi concetti sono presenti in vari linguaggi di programmazione.

Ben-Ari, *Principles of Concurrent Programming*

Questo libretto è completamente dedicato ai problemi della comunicazione tra processi. Ci sono capitoli sulla mutua esclusione, i semafori, i monitor e, fra gli altri, il problema dei filosofi.

Silberschatz et al., *Operating System Concepts*

I Capitoli dal 4 al 6 descrivono i processi e la comunicazione fra processi, tra cui la schedulazione, le sezioni critiche, i semafori, i monitor e i problemi classici di comunicazione fra processi.

### 13.1.3 Deadlock

Coffman et al., "System Deadlocks"

Una breve introduzione ai deadlock, quello che causano, e come possono essere prevenuti o rilevati.

Holt, "Some Deadlock Properties of Computer Systems"

Una discussione sui deadlock. Holt introduce un modello a grafo diretto che si può utilizzare per analizzare alcune situazioni di deadlock.

Isloor e Marsland, "The Deadlock Problem: An Overview"

Un tutorial sui deadlock, che pone una particolare enfasi su sistemi di basi dati. Sono descritti vari modelli ed algoritmi.

### 13.1.4 Gestione della memoria

Denning, "Virtual Memory"

Un articolo classico sui molti aspetti della memoria virtuale. Denning è uno dei pionieri in questo campo ed è stato l'inventore del concetto di insieme di lavoro.

Denning, "Working Sets Past and Present"

Una buona rassegna dei numerosi algoritmi di gestione della memoria e di paginazione. Comprende un'ampia bibliografia.

Knuth, *The Art of Computer Programming*, Vol. 1

In questo libro vengono discussi e confrontati, tra gli altri, gli algoritmi first fit e best fit per la gestione della memoria.

Silberschatz et al., *Applied Operating System Concepts*

I Capitoli 9 e 10 si occupano della gestione della memoria, tra cui lo swapping, la paginazione e la segmentazione. Si accenna a molti algoritmi di paginazione.

### 13.1.5 Input/Output

Chen et al., "RAID: High Performance Reliable Secondary Storage"

L'uso di drive multipli in parallelo per un veloce I/O è una tendenza dei sistemi di fascia alta. Gli autori discutono questa idea ed esaminano differenti organizzazioni in termini di prestazioni, costi ed affidabilità.

*Computer*, Marzo 1994

Questo numero di *Computer* contiene otto articoli su I/O avanzato, simulazione, memoria ad alte prestazioni, cache, I/O per computer paralleli, e multimedia.

Geist e Daniel, "A Continuum of Disk Scheduling Algorithms"

Viene presentato un algoritmo generalizzato per la schedulazione della testina del disco. Viene data un'ampia simulazione e i risultati sperimentali.

Gibson e Van Meter, "Network Attached Storage"

Internet sta portando al bisogno di un vasto spazio di memorizzazione per Web, data-

base e altri server. Questa situazione ha prodotto vari progetti in cui una memorizzazione a sé stante può essere connessa ad una rete. In questo libro, sono discusse diverse architetture che realizzano questo obiettivo.

#### **Ng, "Advances in Disk Technology: Performance Issues"**

I vari fattori che influenzano le prestazioni del disco, come densità lineare dei bit, giri al minuto, numero di testine, numero di settori/tracce possono essere trattati in vari modi per influire sulle prestazioni del sistema. Questi problemi ed il loro impatto sulle prestazioni, sono trattati in questo libro sulla tecnologia del disco, facile da leggere.

#### **Ruempler e Wilkes, "An Introduction to Disk Drive Modeling"**

La prima parte di questo articolo tratta dei moderni drive dei dischi e come funzionano al loro interno, trattando argomenti come la ricerca, la divisione in zone, lo skew delle tracce, i settori di riserva, la cache, la lettura in avanti, e molto di più. La seconda parte parla di come modellare i drive dei dischi.

#### **Walker e Cragon, "Interrupt Processing in Concurrent Processor"**

Implementare interruzioni precise sui supercomputer è una grande sfida. Il trucco è serializzare lo stato e farlo velocemente. Sono discussi molti problemi di progettazione e soluzioni di compromesso.

### **13.1.6 File system**

#### **Harbron, *File Systems***

Un libro sul progetto, le applicazioni e le prestazioni dei file system. Vengono descritte la struttura e gli algoritmi.

#### **McKusick et al., "A Fast File System for UNIX"**

Il file system di UNIX è stato completamente implementato di nuovo per 4.2 BSD. Questo articolo descrive il progetto del nuovo file system, con particolare attenzione agli aspetti legati alle prestazioni.

#### **Silberschatz et al., *Applied Operating System Concepts***

Il Capitolo 11 riguarda il file system. Descrive, fra gli altri argomenti, le operazioni sui file, i metodi di accesso, le directory e l'implementazione.

#### **Stallings, *Operating Systems*. Edizione italiana, *Sistemi operativi*, Jackson Libri 2000.**

Contiene molto materiale circa gli ambienti di sicurezza, in special modo hacker, virus, ed altre minacce.

### **13.1.7 Sistemi operativi multimediali**

#### **ACM Computing Surveys, Dicembre 1995**

Questo numero di ACM Computing Surveys contiene 21 brevi articoli su vari aspetti del multimediale, spaziando da problemi tecnici di basso livello, fino a problemi di applicazioni di alto livello.

#### **Computer, Maggio 1995**

L'argomento di questo numero di Computer è il multimediale e contiene sei articoli su questo argomento. Dopo una breve introduzione, questa rivista parla di televisio-

ne interattiva, server multimediali, gestione della ricerca ed applicazioni mediche e didattiche.

**Lee, "Parallel Video Servers: A Tutorial"**

Molte società vogliono offrire video on demand, e ciò genera la necessità di avere server video in parallelo, su vasta scala, e tolleranti ai guasti. Il problema principale di come costruirli viene trattato qui, comprendendo architettura dei server, smontaggio, politiche di posizionamento, bilanciamento del carico, ridondanza, protocolli e sincronizzazione.

**Leslie et al., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications"**

Molti tentativi di implementare sistemi multimediali si sono basati sull'aggiunta di caratteristiche al sistema operativo esistente. Un approccio alternativo consiste nel cominciare tutto di nuovo dall'inizio, come qui descritto, e costruire da zero un nuovo sistema operativo per il multimediale, senza la necessità di essere compatibili con niente di precedente. Il risultato è un progetto abbastanza differente dai sistemi convenzionali.

**Reddy, "I/O Issues in Multimedia System"**

Quando la gente parla di prestazioni del computer, intende di solito prestazioni della CPU. Tuttavia, per il multimediale, le prestazioni dell'I/O sono parimenti importanti, e questo è l'argomento di questo libro. Tra gli argomenti trattati vi sono lo scheduling del disco, l'utilizzo dello spazio del buffer, ed il controllo degli accessi.

**Sitaram e Dan, "Multimedia Servers"**

I server multimediali sono molto differenti rispetto ai file server tradizionali. Gli autori descrivono in dettaglio le differenze, specialmente nell'area dello scheduling, del sottosistema di memorizzazione e della cache.

### 13.1.8 Sistemi a processori multipli

**Ahmad, "Gigantic Clusters: Where are they and What Are They Doing?"**

Se si vuole avere un'idea dello stato dell'arte dei grandi multicomputer, questo è il libro adatto. Questo libro descrive tale idea e fornisce una visione generale di alcuni dei più grandi sistemi attualmente in funzione. Dato il funzionamento della legge di Moore, è ragionevole scommettere che la dimensione qui citata raddoppierà circa ogni due anni.

**Bhoedjang et al., "User-Level Network Interface Protocols"**

Moltissimi multicomputer stanno inserendo le schede di interfaccia di rete nello spazio riservato all'utente, per migliorare le prestazioni. Agire in questo modo porta molti problemi nella progettazione, ed undici di questi sono qui discussi. Vengono anche messi a confronto molti dei sistemi attuali.

**Computer, Dic. 1996**

Questo numero di *Computer* contiene otto articoli sui multiprocessori. Uno di questi è un tutorial sulla semantica della memoria condivisa, mentre gli altri sette trattano delle applicazioni e delle prestazioni dei multiprocessori.

**Dubois et al., "Synchronization, Coherence, and Event Ordering in Multiprocessors"**

Un tutorial sulla sincronizzazione dei sistemi multiprocessori a memoria condivisa.

Tuttavia alcune idee sono ugualmente applicabili ai sistemi a processore singolo oppure con memoria distribuita.

**Kwok e Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"**

La schedulazione ottima dei job di un multicomputer o multiprocessore è possibile se si conoscono in precedenza tutte le caratteristiche dei job. Il problema è che la schedulazione ottima richiede troppo tempo di calcolo. In questo libro l'autore discute e confronta 27 algoritmi noti per affrontare questo problema in diversi modi.

**Langendoen et al., "Models for Asynchronous Message Handling"**

Una buona prestazione nei multicomputer dipende particolarmente dalla prestazione del sistema di scambio di messaggi, specialmente per come vengono gestiti i messaggi. Le possibilità principali sono i messaggi attivi, le upcall, ed i thread pop-up. Gli autori li descrivono tutti e tre in questo articolo, e poi mostrano i risultati sperimentali, mettendoli a confronto sulla stessa piattaforma hardware.

**Protic et al., "Distributed Shared Memory: Concepts and Systems"**

Questa collezione di 28 articoli precedentemente pubblicati è un buon punto di partenza per un'introduzione alla memoria condivisa distribuita. Comprende molti articoli classici su modelli, algoritmi, ed implementazioni in un singolo e conveniente libro.

**Staenstrom et al., "Trends in Shared Memory Multiprocessing"**

Dove stanno andando i multiprocessori? L'autore ritiene che il futuro stia più nei processori su piccola scala che in quelli su larga scala. Vengono anche discussi modelli, architetture e software parallelo.

**Waldo, "Alive and Well: Jini Technology Today"**

Questo libro è un buon inizio per una breve introduzione a Jini, le sue componenti, e come si adattino tra loro. Forse è indicativo di come verrà diffusa l'informazione in futuro, perché non ha una bibliografia di libri pubblicati, invece fornisce gli URL di molte risorse del Web che trattano di Jini.

### 13.1.9 Sicurezza

*Computer*, Febbraio 2000

Il tema di questo numero di *Computer* è la biometria, con sei articoli su tale argomento. Questi spaziano da un'introduzione all'argomento, attraverso varie tecnologie specifiche, fino ad articoli che trattano questioni di legalità e privacy.

**Denning, "The United State vs Craig Neidorf"**

Quando un giovane hacker ha scoperto e pubblicato informazioni su come funzionasse il sistema telefonico, fu accusato di frode informatica. Questo articolo descrive il caso, che coinvolge molte questioni fondamentali, inclusa la libertà di parola. L'articolo è seguito da opinioni che dissentono e da una confutazione di Denning.

**Denning, *Information Warfare and Security***

Le informazioni sono diventate armi da guerra, sia per i militari che per le aziende. I partecipanti non solo cercano di attaccare i sistemi informativi dell'avversario, ma salvaguardano il loro stesso sistema. In questo affascinante libro, l'autore parla di ogni argo-

mento concepibile relativamente alle strategie di attacco e di difesa, dal furto di dati al packet sniffer. Un libro da leggere per chiunque sia interessato alla sicurezza dei computer.

#### Hafner e Markoff, *Cyberpunk*

Tre irresistibili racconti di giovani hacker che penetrano computer in tutto il mondo vengono qui raccontati dal reporter informatico del *New York Times* che svelò la storia del worm di Internet (Markoff). *Computer*, Febbraio 2000.

#### Johnson e Jajodia, "Exploring Steganografy: Seeing the Unseen"

La steganografia ha una lunga storia, a partire dai giorni in cui chi scriveva rasava la testa del messaggero, scriveva un messaggio sulla testa rasata, e lo spediva dopo che gli erano ricresciuti i capelli. Sebbene le tecniche attuali siano spesso pelose, sono anche digitali. Questo libro è un buon punto di partenza per una introduzione approfondita all'argomento e a come sia attualmente praticato.

#### Ludwig, *The Giant Black Book of Computer Viruses*, 2 ed.

Se volete scrivere un software antivirus e avete bisogno di capire come funzionino i virus fino al livello dei bit, questo è il libro che fa per voi. È discusso estesamente ogni genere di virus, e viene anche fornito su un floppy disk il vero codice della maggior parte di questi. Tuttavia è necessaria una profonda conoscenza della programmazione dei Pentium in assembler.

#### Milojicic, "Security and Privacy"

La sicurezza ha molte sfaccettature, che comprendono sistemi operativi, reti, implicazioni per la privacy e così via. In questo libro sei esperti di sicurezza dicono la loro su tale argomento.

#### Nachenberg, "Computer Virus-Antivirus Coevolution"

Non appena gli sviluppatori di antivirus trovano un modo per rilevare e neutralizzare alcune classi di virus per computer, i creatori di virus fanno un passo avanti e migliorano il virus. Viene qui discusso il gioco tipo gatto col topo, giocato tra virus e antivirus. L'autore non è ottimista riguardo alla vittoria degli antivirus sui virus, il che non è una buona notizia per gli utenti.

#### Pfleeger, *Security in Computing*, 2 ed.

Sebbene siano stati pubblicati molti libri sulla sicurezza dei computer, la maggior parte di essi tratta solo della sicurezza delle reti. Anche questo libro fa ciò, ma ha anche tre capitoli sulla sicurezza dei sistemi operativi, che lo rende un buon punto di riferimento per una maggiore conoscenza dell'argomento.

### 13.1.10 UNIX e Linux

#### Bovet e Cesati, *Understanding the Linux Kernel*

Probabilmente questo libro fornisce la migliore discussione nell'insieme del kernel di Linux. Questo libro parla di processi, gestione della memoria, file system, segnali e molto di più.

#### IEEE, *Information Technology-Portable Operating Systems Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Questo è lo standard. Alcune parti sono effettivamente abbastanza leggibili, specialmente l'Allegato B, "Rationale and Notes", che spesso illumina sul perché certe cose siano fatte in un certo modo. Un vantaggio del fare riferimento a documenti standard sta nel fatto che, per definizione, non ci sono errori. Se si commette un errore tipografico nel nome di una macro durante il processo di editing, non è più un errore, ma diviene ufficiale.

#### Lewine, *POSIX Programmer's Guide*

Questo libro descrive lo standard POSIX molto più nel dettaglio che i documenti standard, ed include discussioni su come convertire vecchi programmi in POSIX e come sviluppare nuovi programmi per POSIX, con esempi.

#### Maxwell, *Linux Core Kernel Commentary*

Le prime 400 pagine di questo libro contengono un sottoinsieme del codice del kernel di Linux. Le ultime 150 contengono commenti al codice, nello stile dei classici libri di John Lions (1966). Se volete capire il kernel Linux nei suoi minimi dettagli, questo è il libro che fa per voi, ma fate attenzione: leggere 40000 linee di codice C, non è cosa da tutti.

#### McKusick et al., *The Design and Implementation of the 4.4BSD Operating System*

Il titolo dice tutto. Ma in realtà il libro è più generale, in quanto tutti i sistemi UNIX sono abbastanza simili internamente. Questo libro è un eccellente punto di riferimento per chiunque voglia imparare qualcosa sul funzionamento interno di UNIX in generale.

### 13.1.11 Windows 2000

#### Cusumano e Selby, "How Microsoft Builds Software"

Vi siete mai domandati come qualcuno possa scrivere un programma di 29 milioni di righe (come Windows 2000) e come possa funzionare? Per scoprire come costruisce Microsoft, e quali cicli di test siano utilizzati per gestire progetti software così ampi, date un'occhiata a questo libro.

#### Norton et al., *Complete Guide to Windows 2000 Professional*

Se state cercando un libro che parli di come installare ed utilizzare Windows 2000, ma che tratti anche in dettaglio le caratteristiche avanzate, come il registro, i file system FAT e NTFS, ActiveX, DCOM, e reti, questa è una buona scelta. Questo libro si trova a metà tra la massa di libri che dicono dove effettuare un click per ottenere un certo effetto, e il libro di Solomon e Russinovich.

#### Rector e Newcomer, *Win32 Programmer*

Se state cercando uno di quei libri da 1500 pagine che diano un sunto di come scrivere programmi Windows, questo non è un cattivo inizio. Questo libro parla di finestre, dispositivi, output grafici, input da mouse e tastiera, stampa, gestione della memoria, librerie, sincronizzazione e molti altri argomenti. Richiede la conoscenza di C o C++.

#### Solomon e Russinovich, *Inside Windows 2000, 3 ed.*

Se volete imparare ad utilizzare Windows 2000, vi sono centinaia di libri. Se volete invece imparare come funziona al suo interno Windows 2000, questa è la migliore scelta possibile. Questo libro parla di numerosi algoritmi interni e strutture dati, con dettagli tecnici considerevoli. Nessun altro libro va così addentro.

### 13.1.12 Principi di progettazione

Brooks, "The Mythical Man Month: Essays on Software Engineering"

Fred Brooks è stato uno dei progettisti dell'OS 360 della IBM. Imparò sulla propria pelle ciò che funziona e ciò che non funziona. I consigli dati in questo libro spiritoso, divertente ed istruttivo sono validi adesso, così come lo erano 25 anni fa, quando furono scritti per la prima volta.

Cooke et al., "UNIX and Beyond: An Interview with Ken Thompson"

Progettare un sistema operativo è più un'arte che una scienza. Di conseguenza, ascoltare gli esperti di questo campo è un buon metodo per imparare qualcosa sull'argomento. Non ci sono esperti migliori di Ken Thompson, co-progettista di UNIX, Inferno e Plan 9. In questa intervista di ampio respiro, Thompson esprime le sue opinioni circa la nostra provenienza e la nostra destinazione in questo campo.

Corbató, "On Building Systems That Will Fail"

Nella sua conferenza per il Premio Turing, il padre del time sharing affronta molti concetti discussi da Brooks in *The Mythical Man Month*. La sua conclusione è che tutti i sistemi complessi alla fine falliranno, e che per avere qualche possibilità di successo, è assolutamente essenziale evitare la complessità e sforzarsi di ricercare semplicità ed eleganza nel progetto.

Crowley, *Operating Systems: A Design-Oriented Approach*

La maggior parte dei libri di testo sui sistemi operativi descrivono i concetti base (processi, memoria virtuale, ecc.) e con pochi esempi, ma non dicono nulla su come progettare un sistema operativo. Questo libro è l'unico che dedichi quattro capitoli a tale argomento.

Lampson, "Hints for Computer System Design"

Butler Lampson, uno dei migliori progettisti a livello mondiale dei sistemi operativi innovativi, prendendo spunto dai suoi anni di esperienza ha messo assieme molti suggerimenti, idee e linee guida in questo libro, interessante ed istruttivo. Come il libro di Brooks, questo libro è necessario per chiunque aspiri a diventare un progettista di sistemi operativi.

Wirth, "A Plea for Lean Software"

Niklaus Wirth, un progettista di sistemi famoso e di molta esperienza, sostiene la necessità di software chiaro e semplice, basato su pochi e semplici concetti, invece dell'immane confusione di molto del software commerciale. Descrive il suo punto di vista discutendo il suo sistema Oberon, un sistema operativo, orientato alla rete, basato su GUI, che è grande 200 KB, compresi il compilatore Oberon e l'editore di testo.

## 13.2 Bibliografia alfabetica

**ABDEL-HAMID, T., e MADNICK, S.: *Software Project Dynamics: An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.**

**ABRAM-PROFETA, E.L., e SHIN, K.G.: "Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Comp. Syst.*, IEEE, pp. 66-75, 1998.**

- ABUTALEB, A., e LI, V.O.K.**: "Paging Strategy Optimization in Personal Communication Systems Wireless Networks," *Wireless Networks*, vol. 3, pp. 195-204, Agosto 1997.
- ACCETTA, M., BIARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., e YOUNG, M.**: "Mach: A New Kernel Foundation for UNIX Development," *Proc. Estate 1986 USENIX Conf.*, USENIX, pp. 93-112, 1986.
- ADAMS, G.B. III, AGRAWAL, D.R., e SIEGEL, H.J.**: "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *Computer*, vol. 20, pp. 14-27, Giugno 1987.
- ADAMS, A., e SASSE, M.A.**: "Taming the Wolf in Sheep's Clothing," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 101-107, 1999.
- AHMAD, I.**: "Gigantic Clusters: Where Are They and What Are They Doing?", *IEEE Concurrency*, vol. 8, pp. 83-85, Aprile-Giugno 2000.
- ALEXANDROV, A.D., IBEL, M., SCHAUER, K.E., e SCHEIMAN, C.J.**: "UFO: a Personal Global File System Based on User-Level Extensions to the Operating System," *ACM Trans. on Computer Systems*, vol. 16, pp. 207-233, Agosto 1998.
- ALFIERI, R.A.**: "An Efficient Kernel-Based Implementation of POSIX Threads," *Proc. Estate 1994 USENIX Tech. Conf.*, USENIX, pp. 59-72, Giugno 1994.
- ALVAREZ, G.A., BURKHARD, W.A., e CRISTIAN, R.**: "Tolerating Multiple Failures in RAID Architectures with Optimal Storage And Uniform Declustering," *Proc. 24th Int'l Symp. on Computer Architecture*, ACM, pp. 62-72, 1997.
- ANDERSON, T.E.**: "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. on Parallel and Distr. Systems*, vol. 1, pp. 6-16, Gennaio 1990.
- ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, ED., e LEVY, H.M.**: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Febbraio 1992.
- ANDREWS, G.R.**: *Concurrent Programming—Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.
- ANDREWS, G.R., e SCHNEIDER, F.B.**: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, Marzo 1983.
- ARON, M., e DRUSCHEL, P.**: "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 223-246, 1999.
- ARORA, A.S., BLUMOFE, R.D., e PLAXTON, C.G.**: "Thread Scheduling for Multiprogrammed Multiprocessors," *Proc. Tenth Symp. on Parallel Algorithms and Architectures*, ACM, pp. 119-129, 1998.
- BAKER, F. T.**: "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, pp. 1, 1972.

- BAKER-HARVEY, M.**: "ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 131-144, 1999.
- BALA, K., KAASHOEK, M.F., WEIHL, W.**: "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating System Design and Implementation*. USENIX, pp. 243-254, 1994.
- BALLINTIJN, G., VAN STEEN, M., e TANENBAUM, A.S.**: "Scalable Naming in Global Middleware." *Proc. 13th Int'l Conf. on Parallel and Distributed Systems*, ISCA, pp. 624-631, 2000.
- BAYS, C.**: "A Comparison of Next-Fit, First-Fit, and Best-Fit" *Commun. of the ACM*, vol. 20, pp. 191-192, Marzo 1977.
- BECK, M., BOHME, H., DZIADZKA, M., KUNITZ, U., MAGNUS, R., VERWORNER, O.**: *Linux Kernel Internals*, 2 ed., Reading, MA: Addison-Wesley, 1998.
- BELADY, L.A., NELSON, R.A., e SHEDLER, G.S.**: "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Commun. of the ACM*, vol. 12, pp. 349-353, Giugno 1969.
- BELL, D., e LA PADULA, L.**: "Secure Computer Systems: Mathematical Foundations and Model," Technical Report MTR 2547 v2, Mitre Corp., Novembre 1973.
- BEN-ARI, M.**: *Principles of Concurrent Programming*, Upper Saddle River, NJ: Prentice Hall International, 1982.
- BERNHARDT, C., e BIERSACK, E.W.**: "The Server Array: A Scalable Video Server Architecture," in *High-Speed Networking for Multimedia Applications*, Amsterdam: Kluwer Publishers, 1996.
- BERSHAD, B.N., SAVAGE, S., PARDYAK, P., BECKER, D., FIUCZYNKI, M., e SIRER, E.G.**: "Protection is a Software Issue," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 62-65, 1995a.
- BERSHAD, B.N., SAVAGE, S., PARDYAK, P., SIRER, E.G., FIUCZYNKI, M., BECKER, D., CHAMBERS, C., e EGGERS, S.**: "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 267-284, 1995b.
- BHOEDJANG, R.A.E.**: *Communication Architectures for Parallel-Programming Systems*, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2000.
- BHOEDJANG, R.A.E., RUHL, T., e BAL, H.E.**: "User-Level Network Interface Protocols," *Computer*, vol. 31, pp. 53-60, Novembre 1998.
- BHUYAN, L.N., YANG, Q., e AGRAWAL, D.P.**: "Performance of Multiprocessor Interconnection Networks," *Computer*, vol. 22, pp. 25-37, Febbraio 1989.
- BIBA, K.**: "Integrity Considerations for Secure Computer Systems." Technical Report 76-

- 371, U.S. Air Force Electronic Systems Division, 1977.
- BIRRELL, A.D., e NELSON, B.J.**: "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol. 2, pp. 39-59, Febbraio 1984.
- BISDIKIAN, C.C., e PATEL, B.V.**: "Issues on Movie Allocation in Distributed Video-on-Demand Systems," *Proc. Int'l Conf. on Commun.*, IEEE, pp. 250-255, 1995.
- BLAUM, M., BRADY, J., RRUCK, J., e MENON, J.**: "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures," *Proc. 21st Int'l Symp. on Computer Architecture*, ACM, pp. 245-254, 1994.
- BLUMOFER, R.D., e LEISERSON, C.E.**: "Scheduling Multithreaded Computations by Work Stealing," *Proc. 35th Annual Symp. on Foundations of Computer Science*, IEEE, pp. 356-368, Novembre 1994.
- BOEHM, B.W.**: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.
- BOLOSKY, W.J., FITZGERALD, R.P., e DOUCEUR, J.R.**: "Distributed Schedule Management in the Tiger Video Fileserver," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 212-223, 1997.
- BORN, O.**: *Inside the Microsoft Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.
- BOVET, D.P., e CESATI, M.**: *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2000.
- BRANDWEIN, R., KATSEFF, H., MARKOWITZ, R., MORTENSON, R., e ROBINSON, B.**: "Nemesis: Multimedia Information Delivery," *Proc. Second ACM Int'l Conf. on Multimedia*, ACM, pp. 473-481, 1994.
- BRICKER, A., GIEN, M., GUILLEMONT, M., LIPKIS, J., ORR, D., e ROZIER, M.**: "A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility," *Proc. EurOpen Spring '91 Conf.*, EurOpen, pp. 13-32, 1991.
- BRINCH HANSEN, P.**: "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 199-207, Giugno 1975.
- BROOKS, F. P., Jr.**: *The Mythical Man-Month: Essays on Software Engineering*, Reading, MA: Addison-Wesley, 1975.
- BROOKS, F. P., Jr.**: "No Silver Bullet—Essence and Accident in Software Engineering," *Computer*, vol. 20, pp. 10-19, Aprile 1987.
- BROOKS, F. P., Jr.**: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary edition*, Reading, MA: Addison-Wesley, 1995.
- BUCHANAN, M., e CHIEN, A.**: "Coordinated Thread Scheduling for Workstation Clusters Under Windows NT," *The USENIX Windows NT Workshop*, USENIX, pp. 47-??, 1997.

- BUGNION, E., DEVINE, S., GOVIL, K., e ROSENBLUM, M.**: "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. on Computer Systems*, vol. 15, pp. 412 - 447, Novembre 1997.
- BUZZARD, G., JACOBSON, D., MACKEY, M., MAROVICH, S., e WILKES, J.**: "An Implementation of the Hamlyn Sender-Managed Interface Architecture," *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 245-259, Ottobre 1996.
- CANT, C.**: *Writing Windows WDM Device Drivers*, Lawrence, KS: R&D Books, 1999.
- CAO, P., LIN, S. B., VENKATARAMAN, S., e WILKES, J.**: "The TickerTAIP Parallel RAID Architecture," *ACM Trans. on Computer Systems*, vol. 12, pp. 236-269, Agosto 1994.
- CAO, P., FELTEN, E.W., KARLIN, A. R., e LI, K.**: "A Study of Integrated Prefetching and Caching Strategies," *Proc. SIGMETRICS Joint Int'l Conf. on Measurement and Modelling of Computer Systems*, ACM, pp. 188-197, 1995.
- CARLEY, L.R., GANGER, G.R., e NAGLE, D.F.**: "MEMS-Based Integrated Circuit Mass Storage Systems" *Commun. of the ACM*, vol. 43, pp. 73-80, Novembre 2000.
- CARR, R.W., e HENNESSY, J.**: "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp. 87-95, 1981.
- CARRIERO, N., e GELENTER, D.**: "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems*, vol. 4, pp. 110-129, Maggio 1986.
- CARRIERO, N. e GELENTER, D.**: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444-458, Aprile 1989.
- CARTER, J.B., BENNETT, J.K., e ZWAENEPOEL, W.**: "Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems," *ACM Trans. on Computer Systems*, vol. 13, pp. 205-243, Agosto 1995
- CERE, C., e NAVASKY, V.**: *The Experts Speak*. New York: Random House, 1984.
- CHANDRA, A., ADLER, M., GOYAL, P., e SHENOY, P.**: "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 45-58, 2000.
- CHASE, J.S., LEVY, H.M., FEELEY, M.J., e LAZOWSKA, E.D.**: "Sharing and Protection in a Single-Address-Space Operating System," *ACM Trans. on Computer Systems*, vol. 12, pp. 271-307, Novembre 1994.
- CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., e PATTERSON, D.A.**: "RAID: High Performance Reliable Storage," *Comp. Surv.*, vol. 26, pp. 145-185, Giugno 1994.
- CHEN, P.M., NG, W.T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., e LOWELL, D.**: "The Rio File Cache: Surviving Operating System Crashes," *Proc. Seventh Int'l Conf. on*

- Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 74-83, 1996.
- CHEN, S., e THAPAR, M.:** "A Novel Video Layout Strategy for Near-Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Computing and Systems*, IEEE, pp. 37-45, 1997.
- CHEN, S., e TOWSLEY, D.:** "A Performance Evaluation of RAID Architectures," *IEEE Trans. on Computers*, vol. 45, pp. 1116-1130, Ottobre 1996.
- CHERITON, D. R.:** "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Review*, vol. 18, pp. 12-20, Ottobre 1984.
- CHERITON, D.:** "The V Distributed System," *Commun. of the ACM*, vol. 31, pp. 314-333, Marzo 1988.
- CHERVENAK, A., VELLANKI, V., e KURMAS, Z.:** "Protecting File Systems: A Survey of Backup Techniques," *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.
- CHOW, T.C.K., e ABRAHAM, J. A.:** "Load Balancing in Distributed Systems." *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401-412, Luglio 1982.
- CLARK, P.C., e HOFFMAN, L.J.:** "BITS: A Smartcard Protected Operating System," *Commun. of the ACM*, vol. 37, pp. 66-70, Novembre 1994.
- COFFMAN, E. G., ELPHICK, M.J., e SHOSHANI, A.:** "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, Giugno 1971.
- COMER, D.:** *Operating System Design. The Xinu Approach*, Upper Saddle River, NJ: Prentice Hall, 1984.
- COOKE, D., URBAN, J., e HAMILTON, S.:** "Unix and Beyond: An Interview with Ken Thompson," *Computer*, vol. 32, pp. 58-64, Maggio 1999.
- CORBALAN, J., MARTORELL, X., e LABARTA, J.:** "Performance-Driven Processor Allocation," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 59-71, 2000.
- CORBATO, F.J.:** "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, Giugno 1991.
- CORBATO, F.J., MERWIN-DAGGETT, M., e DALEY, R.C.:** "An Experimental Time-Sharing System," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.
- CORBATO, F.J., SALTZER, J.H., e CLINGEN, C.T.:** "MULTICS—The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571-583, 1972.
- CORBATO, F.J., ad VYSSOTSKY, V.A.:** "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.
- COURTOIS, P.J., HEYMANS, F., e PARNAS, D.L.:** "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Ottobre 1971.

- CRANOR, C.D., e PARULKAR, G. M.**: "The UVM Virtual Memory System," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 117-130, 1999.
- CROWLEY, C.**: *Operating Systems: A Design-Oriented Approach*. Chicago: Irwin, 1997.
- CUSUMANO, M.A., e SELBY, R.W.**: "How Microsoft Builds Software," *Commun. of the ACM*, vol. 40, pp. 53-61, Giugno 1997.
- DALEY, R.C., e DENNIS, J.B.**: "Virtual Memory, Process, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, Maggio 1968.
- DAN, A., SITARAM, D., e SHAHABUDDIN, P.**: "Scheduling Policies for an On-Demand Video Server with Batching," *Proc. Second Int'l Conf. on Multimedia*, ACM, pp. 15-23, 1994.
- DANDAMUDI, S.P.**: "Reducing Run Queue Contention in Shared Memory Multiprocessors," *Computer*, vol. 30, pp. 82-89, Marzo 1997.
- DE JONGE, W., KAASHOEK, M.F., e HSIEH, W.C.**: "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 15-28, 1993.
- DENNING, D.**: "A Lattice Model of Secure Information Flow," *Commun. of the ACM*, vol. 19, pp. 236-243, 1976.
- DENNING, D.**: "The United States vs. Craig Neidorf," *Commun. of the ACM*, vol. 34, pp. 22-43, Marzo 1991.
- DENNING, D.**: *Information Warfare and Security*. Reading, MA: Addison-Wesley, 1999.
- DENNING, P.J.**: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.
- DENNING, P.J.**: "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.
- DENNING, P.J.**: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Settembre 1970.
- DENNING, P.J.**: "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Gennaio 1980.
- DENNIS, J.B., e VAN HORN, E. C.**: "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, Marzo 1966.
- DEVARAKONDA, M., KISH, B., e MOHINDRA, A.**: "Recovery in the Calypso File System," *ACM Trans. on Computer Systems*, vol. 14, pp. 287-310, Agosto 1996.
- DIFFIE, W., e HELLMAN, M.E.**: "New Directions in Cryptography," *IEEE Trans. on Information Theory*, vol. IT-22, pp. 644-654, Novembre 1976.

- DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.
- DIJKSTRA, E.W.: "The Structure of THE Multiprogramming System." *Commun. of the ACM*, vol. 11, pp. 341-346, Maggio 1968.
- DUBOIS, M., SCHEURICH, C., e BRIGGS, P.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors." *Computer*, vol. 21, pp. 9-21, Febbraio 1988.
- DOUCEUR, J.R., e BOLOSKY, W.J.: "A Large-Scale Study of File-System Contents," *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 59-70, 1999.
- DRUSCHEL, P., PAI, V.S., e ZWAENEPOEL, W.: "Extensible Systems are Leading OS Research Astray," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 38-42, 1997.
- DUDA, K.J., e CHERITON, D.R.: "Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 261-276, 1999.
- EAGER, D.L., LAZOWSKA, E.D., e ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 662-675, Maggio 1986.
- EAGER, D.L., VERNON, M., e ZAHORJAN, J.: "Optimal and Efficient Merging Schedules for Video-on-Demand Servers," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 199-202, 1999.
- EDLER, J., LIPKIS, J., e SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems," *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp. 1-17, Settembre 1988.
- EGAN, J.I., e TEIXEIRA, T.J.: *Writing a UNIX Device Driver*, 2 ed., New York: John Wiley, 1992.
- EL GAMAL, A.: "A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms," *IEEE Trans. on Information Theory*, vol. IT-31, pp. 469-472, Luglio 1985.
- ELLIS, C.S.: "The Case for Higher-Level Power Management," *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 162-167, 1999.
- ENGLER, D.R., GUPTA, S.K., e KAASHOEK, M.F.: "AVM: Application-Level Virtual Memory," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 72-77, 1995a.
- ENGLER, D., CHELF, B., CHOU, A., HALLEM, S.: "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1-16, 2000.
- ENGLER, D.R., e KAASHOEK, M.F.: "Exterminate All Operating System Abstractions," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 78-83, 1995.

- ENGLER, D.R., KAASHOEK, M.F., e O'TOOLE, J. Jr.**: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. on Operating System Principles*, ACM, pp. 251-266, 1995.
- EVEN, S.**: *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.
- FABRY, R.S.**: "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412, Luglio 1974.
- FEELY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., e THEKKATH, C.A.**: "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 201-212, 1995.
- FERGUSON, D., YEMINI, Y., e NIKOLAOU, C.**: "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems," *Proc. Eighth Int'l Conf on Distributed Computing Systems*. IEEE, pp. 491-499, 1988.
- FEUSTAL, E.A.**: "The Rice Research Computer—A Tagged Architecture," *Proc. AFIPS Conf.*, AFIPS, 1972.
- FLINN, J., e SATYANARAYANAN, M.**: "Energy-Aware Adaptation for Mobile Applications," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 48-63, 1999.
- FLUCKIGER, F.**: *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.
- FORD, B., HIBLER, M., LEPREAU, J., TULLMAN, P., BACK, G., CLAWSON, S.**: "Microkernels Meet Recursive Virtual Machines," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137-151, 1996.
- FORD, B., e SUSARLA, S.**: "CPU Inheritance Scheduling," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 91-105, 1996.
- FORD, B., BACK, O., BENSON, O., LEPREAU, J., LIN, A., SHIVERS, O.**: "The Flux OSkit: A Substrate for Kernel and Language Research" *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 38-51, 1997.
- FOTHERINGHAM, J.**: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Ottobre 1961.
- GAFSI, J., e BIERSACK, E.W.**: "A Novel Replica Placement Strategy for Video Servers," *Proc. Sixth Int'l Workshop on Interactive and Distrib. Multimedia Systems*, ACM, pp. 321-335.
- GEIST, R., e DANIEL, S.**: "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Febbraio 1987.
- GELERNTER, D.**: "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 80-112, Gennaio 1985.
- GHORMLEY, D., PETROU, D., RODRIGUES, S., VAHDAT, A., e ANDERSON, T.E.**:

- "SLIC: An Extensible System for Commodity Operating Systems," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 39-46, 1998.
- GIBSON, G.A., e VAN METER, R.: "Network Attached Storage," *Commun. of the ACM*, vol. 43, pp. 37-45, Novembre 2000.
- GILL, D.S., ZHOU, S., e SANDHU, H.S.: "A Case Study of File System Workload in a Large-Scale Distributed Environment," *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 276-277, 1994.
- GOLDBERG, L.A., GOLDBERG, P.W., PHILLIPS, C.A., e SORKIN, G. B.: "Constructing Virus Phylogenies," *Journal of Algorithms*, vol. 26, pp. 188-208, Gennaio 1998.
- GOLDEN, D., e PECHURA, M.: "The Structure of Microcomputer File Systems," *Commun. of the ACM*, vol. 29, pp. 222-230, Marzo 1986.
- GONG, L.: *Inside Java 2 Platform Security*, Reading, MA: Addison-Wesley, 1999.
- GOODHEART, B., e COX, J.: *The Magic Garden Explained*, Upper Saddle River, NJ: Prentice Hall, 1994.
- GOVIL, K., TEODOSIU, D., HUANG, Y., e ROSENBLUM, M.: "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. 17th Symp. on Operating Systems Principles*, ACM, 1999, pp. 154-169.
- GOYAL, P., GUO, X., e VIN, H.M.: "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 107-121, 1996.
- GRAHAM, R.: "Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T., Settembre 1970.
- GRiffin, J.L., SCHLOSSER, S.W., GANGER, G.R., e NAGLE, D.F.: "Operating System Management of MEMS-based Storage Devices," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87-102, 2000.
- GRIMM, R., e BERSHAD, B.: "Security for Extensible Systems," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 62-66, 1997.
- GRIWODZ, C., BAR, M., e WOLF, L.C.: "Long-Term Movie Popularity Models in Video-on-Demand Systems," *Proc. Fifth Int'l Conf. on Multimedia*, ACM, pp. 349-357, 1997.
- GROPP, W., LUSK, E., e SKJELLM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.
- GROSSMAN, D., e SILVERMAN, H.: "Placement of Records on a Secondary Storage Device to Minimize Access Time," *Journal of the ACM*, vol. 20, pp. 429-438, 1973.
- HAFNER, K., e MARKOFF, J.: *Cyberpunk*, New York: Simon and Schuster, 1991.
- HAND, S.M.: "Self-Paging in the Nemesis Operating System," *Proc. Third Symp. on Ope-*

- rating Systems Design and Implementation* USENIX, pp. 73-86, 1999.
- HARBRON, T.R.**: *File Systems*, Upper Saddle River, NJ: Prentice Hall, 1988.
- HARCHOL-BALTER, M., e DOWNEY, A.B.**: "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*. ACM, pp. 13-24, 1996.
- HARRISON, M.A., RUZZO, W.L., e ULLMAN, J.D.**: "Protection in Operating Systems," *Commun. of the ACM*, vol. 19, pp. 461-471, Agosto 1976.
- HART, J.M.**: *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.
- HARTIG, H., HOHMUTH, M., LIEDTKE, J., e SCHONBERG, S.**: "The Performance of Kernel-Based Systems," *Proc. 16th Symp. on Operating Systems Principles*. ACM, pp. 66-77, 1997.
- HARTMAN, J.H., e OUSTERHOUT, J.K.**: "The Zebra Striped Network File System," *ACM Trans. on Computer Systems*, vol. 13, pp. 274-310, Agosto 1995.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., e WEISER, M.**: "Using Threads in Interactive Systems: A Case Study. *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 94-105, 1993.
- HAVENDER, J.W.**: "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, pp. 74-84, 1968.
- HEBBARD, B., et al.**: "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Review*, vol. 14, pp. 7-20, Gennaio 1980.
- HEIDEMANN, J.S., e POPEK, G.J.**: "File-System Development with Stackable Layers," *ACM Trans. on Computer Systems*, vol. 12, pp. 58-89, Febbraio 1994.
- HEYBEY, A., SULLIVAN, M., ENGLAND, P.**: "Calliope: A Distributed Scalable Multimedia Server," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 75-86, 1996.
- HIPSON, P.D.**: *Mastering Windows 2000 Registry*, Alameda, CA: Sybex, 2000.
- HOARE, C.A.R.**: "Monitors, An Operating System Structuring Concept." *Commun. of the ACM*, vol. 17, pp. 549-557, Ottobre 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Febbraio 1975.
- HOLT, R.C.**: "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179-196, Settembre 1972.
- HONEYMAN, P., ADAMSON, A., COFFMAN, K., JANAKIRAMAN, J., JERDONEK, R., e REES, J.**: "Secure Videoconferencing," *The Seventh USENIX Security Symp.*, USENIX, pp. 123-133, 1998.
- HOWARD, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOOTHAM, R.N., e WEST, M.J.**: "Scale and Performance in a Distri-

- buted File System," *ACM Trans. on Computer Systems*, vol. 6, pp. 55-81, Febbraio 1988.
- HUCK, J., e HAYS, J.:** "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Int'l Symp. on Computer Architecture*, ACM, pp. 39-50, 1993.
- HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., e O'MALLEY, S.:** "Logical vs. Physical File System Backup," *Proc. Third Symp. On Operating Systems Design and Implementation*, USENIX, pp. 239-249, 1999.
- IEEE:** *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, Inc., 1990
- ISLOOR, S.S., e MARSLAND, T.A.:** "The Deadlock Problem: An Overview," *Computer*, vol. 13, pp. 58-78, Settembre 1980.
- ITZKOVITZ, A., e SCHUSTER, A.:** "MultiView and Millipage—Fine-Grain Sharing in Page-Based DSMs," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 215-228, 1999.
- IVENS, K.:** *Optimizing the Windows Registry*, Foster City, CA: IDG Books Worldwide, 1998.
- JOHNSON, K.L., KAASHOEK, M.F., e WALLACH, D.A.:** "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213-226, 1995.
- JOHNSON, N. F., e JAJODIA, S.:** "Exploring Steganography: Seeing the Unseen," *Computer*, vol. 31, pp. 26-34, Febbraio 1998.
- JONES, M. B., ROSU, D., e ROSU, M.-C.:** "CPU Reservations and Time Constraints Efficient, Predictable Scheduling of Independent Activities," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 198-211, 1997.
- KAASHOEK, M.F., ENGLER, D. R., GANGER, G.R., BRICENO, H., HUNT, R., MAZIERES, O., PINCKNEY, T., GRIMM, R., JANNOTTI, J., e MACKENZIE, K.:** "Application Performance and Flexibility on Exokernel Systems," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 52-65, 1997.
- KABAY, M.:** "Flashes from the Past." *Information Security*, p. 17, 1997.
- KALLAHALLA, M., e VARMAN, P.J.:** "Optimal Read-Once Parallel Disk Scheduling," *Proc. Sixth Workshop on I/O in Parallel and Distributed Systems*, ACM, pp. 68-77, 1999.
- KARACALI, B., TAI, K.C., e VOUK, M.A.:** "Deadlock Detection of EFSMs Using Simultaneous Reachability Analysis," *Proc. Int'l Conference on Dependable Systems and Networks (DSN 2000)*, IEEE, pp. 315-324, 2000.
- KARLIN, A.R., LI, K., MANASSE, M.S., e OWICKI, S.:** "Empirical Studies of Competi-

- tive Spinning for a Shared-Memory Multiprocessor," Proc. 13th Symp. on Operating Systems Principles, ACM, pp. 41-54, 1991.
- KARLIN, A.R., MANASSE, M.S., McGEOCH, L., e OWICKI, S.: "Competitive Randomized Algorithms for Non-Uniform Problems," Proc. First Annual ACM Symp. on Discrete Algorithms, ACM, pp. 301-309, 1989.
- KARPOVICH, J.F., GRIMSHAW, A.S., e FRENCH, J.C.: "Extensible File System (ELFS): An Object-Oriented Approach to High Performance File I/O," Proc. Ninth Annual Conf. on Object-Oriented Programming Systems, Language, and Applications, ACM, pp. 191-204, 1994.
- KATCHER, D.I., KETTLER, K.A., e STROSNIDER, J.K.: "Real-Time Operating Systems for Multimedia Processing," Proc. Hot Topics in Operating Systems V, IEEE, 1995.
- KAUFMAN, C., PERLMAN, R., e SPECINER, M.: *Network Security*. Upper Saddle River, NJ: Prentice Hall, 1995.
- KAJI, K., BROWNE, J.C., e TRIPATHI, A.: "Computer Systems Research: The Pressure is On," *Computer*, vol. 32, pp. 30-39, Jan 1999.
- KELEHER, P., COX, A., DWARKADAS, S., e ZWAENEPOEL, W.: "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," Proc. USENIX Winter 1994 Conf., USENIX, pp. 115-132, 1994.
- KERNIGHAN, B.W., e PIKE, R.: *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.
- KHALIDI, Y.A., e NELSON, M.N.: "Extensible File Systems in Spring." Proc. 14th Symp. on Operating Systems Principles, ACM, pp. 1-14, 1993.
- KLEIN, D.V.: "Foiling the Cracker: A Survey of, and Improvements to, Password Security," Proc. UNIX Security Workshop II, USENIX, Estate 1990.
- KLEINROCK, L.: *Queueing Systems*. Vol. 1, New York: John Wiley, 1975.
- KLIN, R.L., e GLINERT, E. P.: "Improving GUI Accessibility for People with Low Vision," Proc. Conf. on Human Factors in Computing Systems, ACM, pp. 114-121, 1995.
- KNUTH, D.E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2 Ed., Reading, MA: Addison-Wesley, 1973.
- KOCHAN, S.G., e WOOD, P.H.: *UNIX Shell Programming*, Indianapolis, IN: Hayden Books, 1990.
- KRAVETS, R., e KRISHNAN, P.: "Power Management Techniques for Mobile Communication," Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking, ACM/IEEE, pp. 157-168, 1998.
- KRISHNAN, R.: "Timeshared Video-on-Demand: A Workable Solution," IEEE Multimedia, vol. 6, Gennaio-Marzo 1999, pp. 77-79.

- KRUEGER, P., LAI, T.-H., e DIXIT-RADIYA, V.A.**: "Job Scheduling is More Important than Processor Allocation for Hypercube Computers" *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, pp. 488-497, Maggio 1994.
- KUMAR, V.P., e REDDY, S.M.**: "Augmented Shuffle-Exchange Multistage Interconnection Networks," *Computer*, vol. 20, pp. 30-40, Giugno 1987.
- KWOK, Y.-K., AHMAD, I.**: "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *Computing Surveys*, vol. 31, pp. 406-471, Dicembre 1999.
- LAMPORT, L.**: "Password Authentication with Insecure Communication," *Commun. of the ACM*, vol. 24, pp. 770-772, Novembre 1981.
- LAMPSON, B.W.**: "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, Maggio 1968.
- LAMPSON, B.W.**: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Ottobre 1973.
- LAMPSON, B.W.**: "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Gennaio 1984.
- LAMPSON, B.W., e STURGIS, H.E.**: "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center Technical Report, Giugno 1979.
- LANDWEHR, C. E.**: "Formal Models of Computer Security," *Computing Surveys*, vol. 13, pp. 247-278, Settembre 1981.
- LANGENDOEN, K., BHOEDJANG, R., e BAL, H.E.**: "Models for Asynchronous Message Passing," *IEEE Concurrency*, vol. 5, pp. 28-37, Aprile-Giugno 1997.
- LEBECK, A.R., FAN, X., ZENG, H., ELLIS, C.S.**: "Power Aware Page Allocation," *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, 2000.
- LEE, J.Y.B.**: "Parallel Video Servers: A Tutorial," *IEEE Multimedia*, vol. 5, pp. 20-28, Aprile-Giugno 1998.
- LEE, W., SU, D., WIJESEKERA, D., SRIVASTAVA, J., KENCHAMMANA-HOSEKOTE, D., e FORESTI, M.**: "Experimental Evaluation of PFS Continuous Media File System," *Proc. Sixth Int'l Conf. on Information and Knowledge Management*, ACM, pp. 246-253, 1997.
- LESLIE, I., McAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., e HYDEN, E.**: "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Commun.*, vol. 14, pp. 1280-1297, Luglio 1996.
- LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., e WULF, W.A.**: "Policy/Mechanism Separation in Hydra," *Proc. Fifth Symp. on Operating Systems Principles*, ACM, pp. 132-140, 1975.

- LEWINE, D.**: *POSIX Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.
- LI, K.**: "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Yale Univ., 1986.
- LI, K., e HUDAQ, P.**: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Novembre 1989.
- LI, K., KUMPE, R., HORTON, P., e ANDERSON, T.**: "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," *Proc 1994 Winter Conf.*, USENIX, pp. 279-291, 1994.
- LIEDTKE, J.**: "Improving IPC by Kernel Design," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 175-188, 1993.
- LIEDTKE, J.**: "On Micro-Kernel Construction," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 237-250, 1995.
- LIEDTKE, J.**: "Toward Real Microkernels," *Commun. of the ACM*, vol. 39, pp. 70-77, Settembre 1996.
- LINDE, R.R.**: "Operating System Penetration," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361-368, 1975.
- LIONS, J.**: *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.
- LIU, C. L., e LAYLAND, J.W.**: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46-61, Gennaio 1973.
- LO, V.M.**: "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 30-39, 1984.
- LORCH, J. R., e SMITH, A.J.**: "Reducing Processor Power Consumption by Improving Processor Time Management In a Single-User Operating System," *Proc. Second Int'l Conf. on Mobile Computing and Networking*, ACM, pp. 143-154, 1996.
- LORCH, J., e SMITH, A.J.**: "Apple Macintosh's Energy Consumption," *IEEE Micro*, vol. 18, pp. 54-63, Novembre/Dicembre 1998.
- LOUGHER, P., PEGLER, D., e SHEPHERD, D.**: "Scalable Storage Servers for Digital Audio and Video," *Proc. IEE Int'l Conf. on Storage and Recording Systems*, London: IEE, pp. 140-143, 1994.
- LU, Y.-H., SIMUNIC, I., DE MICHELI, G.**: "Software Controlled Power Management," *Proc. Seventh Int'l Workshop on Hardware/Software Codesign*, ACM, pp. 157-161, 1999.
- LUDWIG, M. A.**: *The Giant Black Book of Computer Viruses*, 2 ed., Show Low, AZ: American Eagle Publications, 1998.
- LUMB, C.R., SCHINDLER, J., GANGER, G.R., e NAGLE, D.E.**: "Towards Higher Disk

- Head Utilization: Extracting Free Bandwidth from Busy Disk Drives,"** *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87-102, 2000.
- MAEKAWA, M., OLDEHOEFT, A. E., e OLDEHOEFT, R.R.: *Operating Systems: Advanced Concepts***, Menlo Park, CA: Benjamin/Cummings, 1987.
- MALKEWITZ, R.: "Head Pointing and Speech Control as a hands-free interface to Desktop Computing,"** *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 182-188, 1998.
- MANARIS, B., e HARKREADER, A.: "SUITEKeys: A Speech Understanding Interface for the Motor-Control Challenged,"** *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 108-115, 1998.
- MARK, A.R.: "The Development of Destination-Specific Biometric Authentication,"** *Proc. Tenth Conf. on Computers, Freedom and Privacy*, ACM, pp. 77-80, 2000.
- MARKOWITZ, J.A.: "Voice Biometrics,"** *Commun. of the ACM*, vol. 43, pp. 66-73, Settembre 2000.
- MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., e MARKATOS, E.P.: "First-Class User-Level Threads,"** *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 110-121, 1991.
- MATTHEWS, J.N., ROSELLI, D., COSTELLO, A.M., WANG, R.Y., e ANDERSON, T.E.: "Improving the Performance of Log-Structured File Systems with Adaptive Methods,"** *Proc. 16th Symp. on Operating Systems Prin.*, ACM, pp. 238-251, 1997.
- MAXWELL, S. E.: *Linux Core Kernel Commentary***, Scottsdale, AZ: Coriolis, 1999.
- MAZIERES, D., KAMINSKY, M., KAASHOEK, M.F., e WITCHEL, E.: "Separating Key Management from File System Security,"** *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 124-139, 1999.
- McDANIEL, T.: "Magneto-Optical Data Storage,"** *Commun. of the ACM*, vol. 43, pp. 57-63, Novembre 2000.
- McKUSICK, M.K., BOSTIC, K., KARELS, M.J., e QUARTERMAN, J.S.: *The Design and Implementation of the 4.4 BSD Operating System***, Reading, MA: Addison-Wesley, 1996.
- McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., e FABRY, R.S.: "A Fast File System for UNIX,"** *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Agosto 1984.
- MEDINETS, D.: *UNIX Shell Programming Tools***, New York, NY: McGraw-Hill, 1999.
- MELLOR-CRUMMEV, J.M., e SCOTT, M. L.: "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,"** *ACM Trans. on Computer Systems*, vol. 9, pp. 21-65, Febbraio 1991.
- MERCER, C.W.: "Operating System Support for Multimedia Applications,"** *Proc. Second Int'l Conf. on Multimedia*, ACM, pp. 492-493, 1994.

- MILLER, F. W.**: "pk: A POSIX Threads Kernel," *FREENIX Track: USENIX Annual Technical Conference*, USENIX, pp. 179-182, 1999.
- MILOJICIC, D.**: "Operating Systems: Now and in the Future," *IEEE Concurrency*, vol. 7, pp. 12-21, Gennaio-Marzo 1999.
- MILOJICIC, D.**: "Security and Privacy," *IEEE Concurrency*, vol. 8, pp. 70-79, Aprile-Giugno 2000.
- MONROSE, F. e RUBIN, A.**: "Authentication Via Keystroke Dynamics," *Proc. Conf. on Computer and Communications Security*, ACM, pp. 48-56, 1997.
- MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S. e SMITH, ED.**: "Andrew: A Distributed Personal Computing Environment," *Commun. of the ACM*, vol. 29, pp. 184-201, Marzo 1986.
- MORRIS, R. e THOMPSON, K.**: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Novembre 1979.
- MULLENDER, S.J. e TANENBAUM, A.S.**: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365-368, Aprile 1984.
- MYERS, A. C. e LISKOV, B.**: "A Decentralized Model for Information Flow Control," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 129-142, 1997.
- NACHENBERG, C.**: "Computer Virus-Antivirus Coevolution" *Commun. of the ACM*, vol. 40, pp. 46-51, Gennaio 1997.
- NEMETH, E., SNYDER, G., SEEBASS, S., e HEIN, T.R.**: *UNIX System Administration Handbook*, 2 ed., Upper Saddle River, NJ: Prentice Hall, 2000.
- NEWHAM, C. e ROSENBLATT, B.**: *Learning the Bash Shell*, Sebastopol, CA: O'Reilly & Associates, 1998.
- NEWTON, G.**: "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *Operating Systems Review*, vol. 13, pp. 33-44, Aprile 1979.
- NIEH, J. e LAM, M. S.**: "The Design, Implementation and Evaluation of SMART a Scheduler for Multimedia Applications," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 184-197, 1997.
- NIST (National Institute of Standards e Technology)**: FIPS Pub. 180-1, 1995.
- NG, S. W.**: "Advances in Disk Technology: Performance Issues," *Computer*, vol. 31, pp. 75-81, Maggio 1998.
- NORTON, P., MUELLER, J. e MANSFIELD, R.**: *Complete Guide to Windows 2000 Professional*, Indianapolis, IN: Sams, 2000.
- OKI, B., PFLUEGL, M., SIEGEL, A., e SKEEN, D.**: "The Information Bus—An Archi-

- lecture for Extensible Distributed Systems," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 58-68, 1993.
- ONEY, W.**: *Programming the Microsoft Windows Driver Model*, Redmond, WA: Microsoft Press, 1999.
- ORGANICK, E.I.**: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.
- ORLOV, S. S.**: "Volume Holographic Data Storage," *Commun. of the ACM*, vol. 43, pp. 47-54, Novembre 2000.
- OUSTERHOUT, J.K.**: "Scheduling Techniques for Concurrent Systems," *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22-30, 1982.
- PAI, V.S., DRUSCHEL, P., e ZWAENEPOEL, W.**: "IO-Lite: A Unified I/O Buffering and Caching System," *ACM Trans. on Computer Systems*, vol. 18, pp. 37-66, Febbraio 2000.
- PAKIN, S., KARAMCHETI, V., CHIEN, A.A.**: "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors," *IEEE Concurrency*, vol. 5, pp. 60-73, Aprile-Giugno 1997.
- PANKANTI, S., BOLLE, R.M., e JAIN, A.**: "Biometrics: The Future of Identification," *Computer*, vol. 33, pp. 46-49, Febbraio 2000.
- PATE, S.D.**: *UNIX Internals A Practical Approach*, Reading, MA: Addison-Wesley, 1996.
- PATTERSON, R.H., GIBSON, G. A., GINTING, E., STODOLSKY, D., e ZELENKA, J.**: "Informed Prefetching and Caching," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 79-95, 1995.
- PATTERSON, D. A., GIBSON, G., e KATZ, R.**: "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109-166, 1988.
- PENTLAND, A., e CHOUDHURY, T.**: "Face Recognition for Smart Environments," *Computer*, vol. 33, pp. 50-55, Febbraio 2000.
- PETERSON, G. L.**: "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, Giugno 1981.
- PETROU, D., MILFORD, J., e GIBSON, G.**: "Implementing Lottery Scheduling," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 1-14, 1999.
- PETZOLD, C.**: *Programming Windows, 5 ed.*, Redmond, WA: Microsoft Press, 1999.
- PFLEIGER, C.P.**: *Security in Computing, 2 ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.
- PHILBIN, J., EDLER, J., ANSHUS, O.J., DOUGLAS, C.C., e LI, K.**: "Thread Scheduling for Cache Locality," *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 60-71, 1996.

- PRECHELT, L.: "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, pp. 23-29, Ottobre 2000.
- PROTIC, J., TOMASEVIC, M., e MILUTINOVIC, V.: *Distributed Shared Memory: Concepts and Systems*, Los Alamitos, CA: IEEE Computer Society, 1998.
- RAWSON, F. L. III: "Experience with the Development of a Microkernel-Based, Multi-Server Operating System," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 2-7, 1997.
- RECTOR, B.E., e NEWCOMER, J.M.: *Win32 Programming*, Reading, MA: Addison-Wesley, 1997.
- REDDY, A.L.N., e WYLLIE, J.C.: "Disk Scheduling in a Multimedia 110 System," *Proc. ACM Multimedia Conf.*, ACM, pp. 225-233, 1992.
- REDDY, A.L.N., e WYLLIE, J.C.: "I/O Issues in a Multimedia System," *Computer*, vol. 27, pp. 69-74, Marzo 1994.
- RITCHIE, D. M.: "Reflections on Software Research," *Commun. of the ACM*, vol. 27, pp. 758-760, Agosto 1984.
- RITCHIE, D. M., e THOMPSON, K.: "The UNIX Timesharing System," *Commun. of the ACM*, vol. 17, pp. 365-375, Luglio 1974.
- RIVEST, R.L.: "The MD5 Message-Digest Algorithm," RFC 1320, Aprile 1992.
- RIVEST, R.L., SHAMIR, A., e ADLEMAN, L.: "On a Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Commun. of the ACM*, vol. 21, pp. 120-126, Febbraio 1978.
- ROBBINS, A.: *UNIX in a Nutshell: A Desktop Quick Reference for SVR4 and Solaris 7*, Sebastopol, CA: O'Reilly & Associates, 1999.
- ROMPOGIANNAKIS, Y., NERJES, G., MUTH, P., PATERAKIS, M., TRIANTAFILLOU, P., e WEIKUM, G.: "Disk Scheduling for Mixed-Media Workloads in a Multimedia Server," *Proc. Sixth Int'l Conf. on Multimedia*, ACM, pp. 297-302, 1998.
- ROSENBLUM, M., e OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. on Oper. Sys. Prin.*, ACM, pp. 1-15, 1991.
- ROSELLI, D., e LORCH, J.R.: "A Comparison of File System Workloads," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 41-54, 2000.
- ROZIER, M., ABBROSSIMOV, V., ARMAND, E., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, E., KAISER, C., LEONARD, P., LANGLOIS, S., e NEUHAUSER, W.: "Chorus Distributed Operating Systems," *Computing Systems*, vol. 1, pp. 305-379, Ottobre 1988.
- RUEMMELER, C., e WILKES, J.: "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, pp. 17-28, Marzo 1994.

- SACKMAN, H., ERIKSON, W.J., e GRANT, E. E.: "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Commun. of the ACM*, vol. 11, pp. 3-11, Gennaio 1968.
- SALTZER, J.H.: "Protection and Control of Information Sharing in MULTICS," *Commun. of the ACM*, vol. 17, pp. 388-402, Luglio 1974.
- SALTZER, J.H., REED, D. P., e CLARK, D. D.: "End-to-End Arguments in System Design," *ACM Trans. on Computer Systems*, vol. 2, pp. 277-277, Novembre 1984.
- SALTZER, J.H., e SCHROEDER, M.D.: "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Settembre 1975.
- SALUS, P. H.: "UNIX At 25," *Byte*, vol. 19, pp. 75-82, Ottobre 1994.
- SANDHU, R. S.: "Lattice-Based Access Control Models," *Computer*, vol. 26, pp. 9-19, Novembre 1993.
- SANTRY, D., FEELEY, M.J., HUTCHINSON, N.C., VEITCH, A. C.: "Elephant: The File System That Never Forgets," *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 2-7, 1999a.
- SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A. C., CARTON, R.W., e OFIR, J.: "Deciding When to Forget in the Elephant File System." *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 110-123, 1999b.
- SATYANARAYANAN, M., HOWARD, J.H., NICHOLS, D.N., SIDEBOOTHAM, R.N., SPECTOR, A. Z., e WEST, M.J.: "The ITC Distributed File System: Principles and Design," *Proc. of the Tenth Symp. on Operating System Prin.*, ACM, pp. 35-50, 1985.
- SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., e ANDERSON, T.: "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." *ACM Trans. on Computer Systems*, vol. 15, pp. 391-411, Novembre 1997.
- SCALES, D. J., e GHARACHORLOO, K.: "Towards Transparent and Efficient Software Distributed Shared Memory," *Proc. 16<sup>th</sup> Symp. on Operating Systems Principles*, ACM, pp. 157-169, 1997.
- SCHMIDT, B.K., LAM, M. S., e NORTHCUTT, J. D.: "The Interactive Performance of Slim: A Stateless, Thin-Client Architecture," *Proc. 17<sup>th</sup> Symp. on Operating Systems Principles*, ACM, pp. 32-47, 1999.
- SCOTT, M., LEBLANC, T., e MARSH, B.: "Multi-model Parallel Programming in Psyche," *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp. 70-78, 1990.
- SEAWRIGHT, L.H., e MACKINNON, R.A.: "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems Journal*, vol. 18, pp. 4-17, 1979.
- SEGARRA, M. -T., e ANDRI, F.: "MFS: a Mobile File System Using Generic System Services," *Proc. 1999 Symp. on Applied Computing*, ACM, pp. 419-420, 1999.

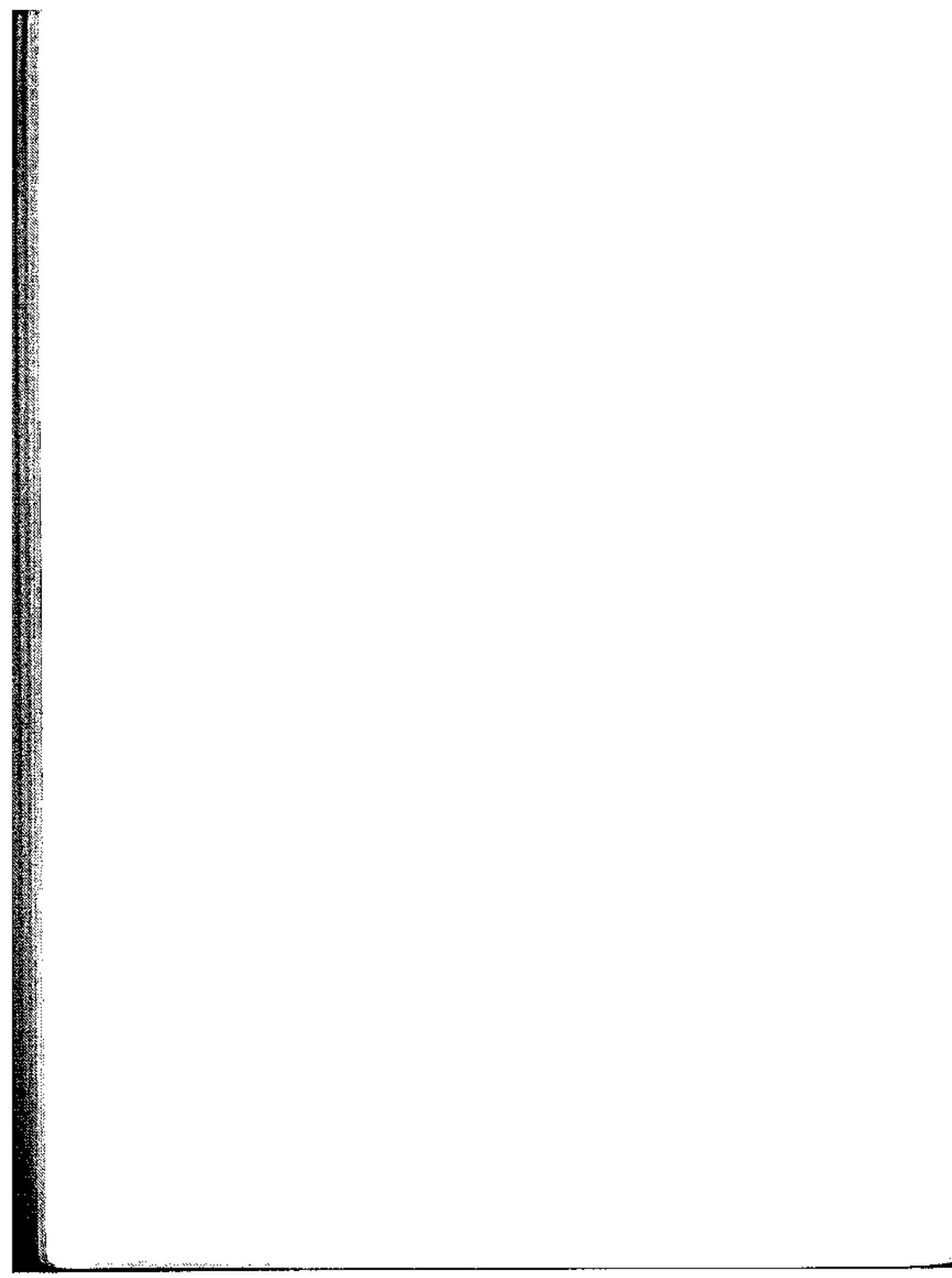
- SELTZER, M., ENDO, Y., SMALL, C., e SMITH, K.**: "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 213-227, 1994.
- SHENOV, P.J., e VIN, H.M.**: "Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers," *Perf. Eval. Journal*, vol. 38, pp. 175-199, 1999.
- SHENOV, P.J., GOYAL, P., e VIN, H.M.**: "Architectural Considerations for Next Generation File Systems," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 457-467, 1999.
- SILBERSCHATZ, A., GALVIN, P.B., e GAGNE, G.**: *Applied Operating System Concepts*, New York: Wiley, 2000.
- SIMON, R. J.**: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.
- SITARAM, D., e DAN, A.**: *Multimedia Servers*, San Francisco, CA: Morgan Kaufman, 2000.
- SLAUGHTER, L., OARD, D.W., WARNICK, V.L., HARDING, J.L., e WILKERSON, G.J.**: "A Graphical Interface for Speech-Based Retrieval," *Proc. Third ACM Conf. on Digital Libraries*, ACM, pp. 305-306, 1998.
- SMALL, C., e SELTZER, M.**: "MiSFIT: constructing Safe Extensible Systems," *IEEE Concurrency*, vol. 6, pp. 34-41, Luglio-Settembre 1998.
- SMITH, D.K., e ALEXANDER, R.C.**: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, New York: William Morrow, 1988.
- SNIR, M., OTTO, S. W., HUSS-LEDERMAN, S., WALKER, D.W., e DONGARRA, J.**: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.
- SOLOMON, D. A., e RUSSINOVICH, M. E.**: *Inside Windows 2000*, 3 ed., Redmond, WA: Microsoft Press, 2000.
- SPAFFORD, E., HEAPHY, K., e FERBRACHE, D.**: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.
- STALLINGS, W.**: *Operating Systems*, 4 Ed., Upper Saddle River, NJ: Prentice Hall, 2001. Edizione italiana, *Sistemi operativi*, Jackson Libri 2000.
- STEEKKISTE, P.A.**: "A Systematic Approach to Host Interface Design for High-Speed Networks," *Computer*, vol. 27, pp. 47-57, Marzo 1994.
- STEINMETZ, R., e NAHRSTEDT, K.**: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.
- STENSTROM, P., HAGERSTEN, E., LILJA, D. J., MARTONOSI, M., e VENUGOPAL, M.**: "Trends in Shared Memory Multiprocessing," *Computer*, vol. 30, pp. 44-50, 1997.
- STETS, IL, DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS,**

- L., PARTHASARATHY, S., e SCOTT, M.: "Cashmere—2L Software Coherent Shared Memory on a Clustered Remote-Write Network," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170-183, 1997.
- STEVENS, W.R.: *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.
- STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, New York: Doubleday, 1989.
- STONE, H. S., e BOKHARI, S.H.: "Control of Distributed Processes," *Computer*, vol. 11, pp. 97-106, Luglio 1978.
- TAI, K. C., e CARVER, R.H.: "VP: A New Operation for Semaphores," *Operating Systems Review*, vol. 30, pp. 5-11, Luglio 1996.
- TALLURI, M., e HILL, M.D.: "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 171-182, 1994.
- TALLURI, M., HILL, M.D., e KHALIDI, Y.A.: "A New Page Table for 64-bit Address Spaces," *Proc. 15th Symp. on Operating Systems Prin.*, ACM, pp. 184-200, 1995.
- TANENBAUM, A. S.: *Operating Systems: Design and Implementation*, Upper Saddle River, NJ: Prentice Hall, 1987.
- TANENBAUM, A. S.: *Computer Networks*, Upper Saddle River, NJ: Prentice Hall, 1996.
- TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G.J., MULLENDER, S.J., JANSEN, J., e VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46-63, Dicembre 1990.
- TANENBAUM, A. S., e VAN STEEN, M.R.: *Distributed Systems*, Upper Saddle River, NJ: Prentice Hall, 2002.
- TANENBAUM, A. S., e WOODHULL, A. S.: *Operating Systems: Design and Implementation*, 2 ed., Upper Saddle River, NJ: Prentice Hall, 1997.
- TAYLOR, R.N., MEDVIDOVIC, N., ANDERSON, K. M., WHITEHEAD, E.J., e RDB-BINS, J.E.: "A Component- and Message-Based Architectural Style for GUI Software," *Proc. 17th Int'l Conf. on Software Engineering*, ACM, pp. 295-304, 1995.
- TEORY, T.J.: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1-11, 1972.
- THEKKATH, C.A., MANN, T., e LEE, E.K.: "Frangipani: A Scalable Distributed File System," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 224-237, 1997.
- THOMPSON, K.: "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761-763, Agosto 1984.

- TRONO, J. A.**: "Comments on Tagged Semaphores." *Operating Systems Review*, vol. 34, pp. 7-11, Ottobre 2000.
- TUCKER, A., e GUPTA, A.**: "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp. 159-166, 1989.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., e BROWN, R.**: "Design Tradeoffs for Software-Managed TLBs," *ACM Trans. on Computer Systems*, vol. 12, pp. 175-205, Agosto 1994.
- VAHALIA, U.**: *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 1996.
- VAN BUSKIRK, R., e LALOMIA, M.**: "A Comparison of Speech and Mouse/Keyboard GUI Navigation," *Conference Companion on Human Factors in Computing Systems*, ACM, p. 96, 1995.
- VAN DOORN, L.**: *The Design and Application of an Extensible Operating System*, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2001.
- VAN DOORN, L., HOMBURG, P., e TANENBAUM, A. S.**: "Paramecium: An Extensible Object-Based Kernel," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 86-89, 1995.
- VAN STEEN, M., HAUCK, F.J., BALLINTIJN, G., e TANENBAUM, A. S.**: "Algorithmic Design of the Globe Wide-Area Location Service," *Computer Journal*, vol. 41, pp. 207-310, 1998a.
- VAN STEEN, M., HAUCK, F.J., HOMBURG, P., e TANENBAUM, A. S.**: "Locating Objects in Wide-Area Systems," *IEEE Communications Magazine*, vol. 36, pp. 104-109, Gennaio 1998b.
- VAN STEEN, M., HOMBURG, P., e TANENBAUM, A. S.**: "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. 7, pp. 70-78, Gennaio-Marzo 1999a.
- VAN STEEN, M., TANENBAUM, A. S., KUZ, I., e SIPS, H.J.**: "A Scalable Middleware Solution for Advanced Wide-Area Web Services," *Distributed Systems Engineering*, vol. 7, pp. 34-42, 1999b.
- VASWANI, R., e ZAHORJAN, J.**: "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 26-40, 1991.
- VENKATASUBRAMANIAN, N., e RAMANATHAN, S.**: "Load Management in Distributed Video Servers," *Proc. 17th Int'l Conf on Distrib. Computing Systems*, IEEE, pp. 528-535, 1997.
- VINOSKI, S.**: "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 35, pp. 46-56, Feb 1997.

- VISCAROLA, P.C., e MASON, W. A.**: *Windows NT Device Driver Development*, Indianapolis, IN: Macmillan Technical Publishing, 1999.
- VOGELS, W.**: "File System Usage in Windows NT 4.0," *Proc. 17<sup>th</sup> Symp. on Operating Systems Principles*, ACM, pp. 93-109, 1999.
- VON EICKEN, T., BASU, A., BUCH, V., e VOGELS, W.**: "U-Net: a User-Level Network Interface for Parallel and Distributed Computing," *Proc. 15<sup>th</sup> Symp. on Operating Systems Principles*, ACM, pp. 40-53, 1995.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S.C., SCHAUER, K.E.**: "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19<sup>th</sup> Int'l Symp. on Computer Architecture*, ACM, pp. 256-266, 1992.
- WAHBE, R., LUCCO, S., ANDERSON, T., e GRAHAM, S.**: "Efficient Software-Based Fault Isolation," *Proc. 14<sup>th</sup> Symp. on Operating Systems Principles*, ACM, pp. 203-216, 1993.
- WALDO, J.**: "The Jini Architecture for Network-Centric Computing," *Commun. of the ACM*, vol. 42, pp. 76-82, Luglio 1999.
- WALDO, J.**: "Alive and Well: Jini Technology Today," *Computer*, vol. 33, pp. 107-109, Giugno 2000.
- WALDSPURGER, C.A., e WEIHL, W.E.**: "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1-12, 1994.
- WALKER, W., e CRAGON, H. G.**: "Interrupt Processing in Concurrent Processors," *Computer*, vol. 28, pp. 36-46, Giugno 1995.
- WAN, G., e LIN, E.**: "A Dynamic Paging Scheme for Wireless Communication Systems," *Proc. Third Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 195-203, 1997.
- WANG, C., GOEBEL, V., e PLAGEMANN, T.**: "Techniques to Increase Disk Access Locality in the Minorca Multimedia File System," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 147-150, 1999.
- WANG, R.Y., ANDERSON, T.E., e PATTERSON, D.A.**: "Virtual Log Based File Systems for a Programmable Disk," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 29-43, 1999.
- WEISER, M., WELCH, B., DEMERS, A., e SHENKER, S.**: "Scheduling for Reduced CPU Energy," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 13-23, 1994.
- WILKES, J., GOLDING, R., STAELIN, C., e SULLIVAN, T.**: "The HP AutoRAID Hierarchical Storage System," *ACM Trans. on Computer Systems*, vol. 14, pp. 108-136, Febbraio 1996.

- WIRTH, N.: "A Plea for Lean Software," *Computer*, vol. 28, pp. 64-68, Febbraio 1995.
- WOLMAN, A., VOELKER, M., SHARMA, N., CARDWELL, N., KARLIN, A., e LEVY, H.M.: "On the Scale and Performance of Cooperative Web Proxy Caching," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 16-31, 1999.
- WONG, C.K.: *Algorithmic Studies in Mass Storage Systems*, New York: Computer Science Press, 1983.
- WONG, P.C., e LEE, Y.B.: "Redundant Array of Inexpensive Servers (RAIS)," *Proc. Int'l Conf. on Commun.*, IEEE, pp. 787-792, 1997.
- WORTHINGTON, B.L., GANGER, G.R., e PATT, Y.N.: "Scheduling Algorithms for Modern Disk Drives," *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, pp. 241-251, 1994.
- WU, M., e SHU, W.: "Scheduling for Large-Scale Parallel Video Servers," *Proc. Sixth Symp. on Frontiers of Massively Parallel Computation*, IEEE, pp. 126-133, 1996.
- WULE, W. A., COHEN, E. S., CORWIN, W.M., JONES, A. K., LEVIN, R., PIERSON, C., e POLLACK, E.J.: "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. of the ACM*, vol. 17, pp. 337-345, Giugno 1974.
- YOUNG, M., TEVANIAN, A., Jr., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., e BARON, R.: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63-76, 1987.
- ZACHARY, G. P.: *Showstopper*, New York: Maxwell MacMillan, 1994.
- ZAHORJAN, J., LAZOWSKA, E. D., e EAGER, D. L.: "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 180-198, Aprile 1991.
- ZEKAUSKAS, M.J., SAWDON, W. A., e BERSHAD, B.N.: "Software Write Detection for a Distributed Shared Memory," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87-100, 1994.
- ZOBEL, D.: "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, vol. 17, pp. 6-16, Ottobre 1983.
- ZUBERI, K. M., PILLAI, P., e SHIN, K. G.: "EMERALDS: A Small-Memory Real-Time Microkernel," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 277-299, 1999.
- ZWICKY, E.D.: "Torture-Testing Backup and Archive Programs: Things You Ought to Know but Probably Would Rather Not," *Proc. Fifth Conf. on Large Installation Systems Admin.*, USENIX, pp. 181-190, 1991.



# INDICE ANALITICO

## A

A chiave simmetrica, 542  
Accesso Diretto in Memoria, 257  
Acknowledgement, 110  
ACL, 596  
Address space, (vedi Spazio di indirizzamento)  
Agenti, 586  
Aging, (vedi Invecchiamento)  
Algoritmi a pila, 211  
Algoritmi di allocazione del processore, 502  
Algoritmi di rimpiazzamento delle pagine, 197  
    dell'orologio, 201  
    FIFO, 199  
    globale, 217  
    locale, 217  
    LRU, 201  
    NFU, 203  
    NRU, 198  
    ottimale, 197  
    PFF, 217  
    seconda opportunità, 200  
    working set, 204  
    WSClock, 208  
Algoritmo a canne d'organo, 452  
Algoritmo buddy, 663  
Algoritmo del banchiere, 162

Algoritmo dello struzzo, 152  
Algoritmo di schedulazione  
    a due livelli, 482  
    a frequenza monotona, 437  
    code multiple, 132  
    con priorità alla scadenza più vicina, 439  
    in tempo reale, 436  
    multicomputer, 502  
    multimediali, 435  
    multiprocessore, 481  
    per affinità, 482  
    primo arrivato, primo servito, 126  
    round robin, 130  
    schedulazione a lotteria, 134  
    schedulazione a tre livelli, 128  
    schedulazione con priorità, 131  
    schedulazione fair-share, 134  
    schedulazione garantita, 133  
    shortest job first, 127  
    shortest job next, 133  
    shortest remaining time next, 128  
Algoritmo di sostituzione delle pagine  
    Linux, 662  
    Windows 2000, 752  
Algoritmo per il controllo dell'ammissione, 425  
Algoritmo scan-EDF, 459  
Allocazione a lista concatenata, 373  
Allocazione contigua, 372

Andrew File System, 519  
 Anelli di protezione, 242  
 Anomalia di Belady, 211  
 Aperiodico, evento, 135  
 API, (*vedi Application Program Interface*)  
 Applet, 586  
 Application Program Interface, 48  
 Architettura, 3  
 Architettura etichettata, 598  
 Assegnazione dei nomi, 801  
 Asymmetric Digital Subscriber Loop, 422  
 Attacco tramite negazione del servizio, 569  
 Attesa attiva, 95, 266  
 Attributi non residenti, 767  
 Autenticazione, 110  
 Autenticazione dell'utente, 544  
 Avversari, 539  
 Avviare Windows 2000, 743  
 Avvio di UNIX, 652  
 Azione atomica, 100

**B**

Backoff esponenziale binario, 479, 509  
 Barriera, 111  
 Basic Input Output System, 30  
 Batch, sistemi, 7  
 Berkeley Fast File System, 684  
 Berkeley Software Distribution, (*vedi BSD*)  
 Best fit, 184  
 Bilanciamento del carico, 502  
     algoritmo basato sull'offerta, 505  
     algoritmo deterministico basato  
         sulla teoria dei grafi, 503  
     algoritmo euristico distribuito  
         iniziato dal mittente, 504  
     algoritmo euristico distribuito  
         iniziato dal ricevente, 504  
 Biometrie, 555  
 BIOS, (*vedi Basic Input Output System*)  
 Bit di attesa della sveglia, 100  
 Bit presente/assente, 188  
 Bit sporco, 193  
 Bitmap, 328  
 Blocco della firma, 544  
 Blocco di avvio, 371  
 Blocco di controllo dei processi, 72  
 Bomba logica, 561  
 Bounded-buffer problem, (*vedi Buffer  
     a capacità limitata, problema del*)  
 Bridge, 509  
 BSD, 12  
 Buffer a capacità limitata, problema del, 98

Buffer cache, 394  
 Bufferizzazione, 274  
 Bufferizzazione doppia, 275  
 Busy waiting, (*vedi Attesa attiva*), 27  
 Byte code, 591

**C**

Cache, 80  
 Cache di buffer, 670  
 Cache hit, 21  
 Cache line, 21  
 Cache-coherent NUMA, 472  
 Call gate, 242  
 Cambio di contesto, 24, 130  
 Canale nascosto, 608  
 Capability, 598  
 Capability list, 598  
 Carattere di escape, 316  
 Caratteri magici, shell, 630  
 Carte a valore immagazzinato, 553  
 Cavallo di troia, 558  
 CD registrabili, 289  
 CD riscrivibili, 290  
 CD-R, (*vedi CD registrabili*)  
 CD-ROM, 285  
     file system, 290  
     multisessione, 290  
     sessioni, 290  
     XA, 290  
 CD-RW, (*vedi CD riscrivibili*)  
 CERT, 586  
 Certificato, 544  
 Challenge-response, 552  
 Check-point, (*vedi Punto di controllo*)  
 Chiamata di Procedura Asincrona, 719  
 Chiamata di Procedura Rinviabile, 719  
 Chiamata di sistema, 21, 40  
     I/O UNIX, 668  
     I/O Win32, 757  
     memoria UNIX, 657  
     processi UNIX, 638  
     sicurezza UNIX, 694  
     thread in UNIX, 641  
     Win32, gestione della memoria, 749  
 Chiamate API Win32, 709  
     gestione dei processi, 734  
     gestione dei thread, 734  
     gestione della memoria, 749  
     gestione I/O, 757  
     sicurezza, 168  
 Chiamate di procedura remote, 496

- Chiamate di sistema, file system di UNDX, 678  
Chiavi, 539  
Cilindro, 21  
Client sottili, 335  
Client stub, 496  
Client X, 331  
Clock, 305  
Cluster di computer, 486  
Clusters of Workstations, 486  
CMOS, 23  
CMS, (*vedi* Conversational Monitor System)  
Codice correttore di errore, 254  
Codice mobile, 587  
Codifica audio, 426  
Codifica video, 428  
Coerenza architetturale, 790  
Collegamento simbolico, (*vedi* Link simbolico)  
Comandi di protezione, 603  
Commutazione di banchi, 749  
Commutazione di circuito, 488  
Commutazione di pacchetto store-and-forward, 488  
Compatible Time Sharing System, 11  
Compattazione della memoria, 181  
Compilatore C portatile, 622  
Compressione di file, 772  
Compressione video, 430  
Compromessi tra spazio e tempo, 812  
Compute-bound, 121  
Computer alimentati a batteria, 824  
Comunicazione fra processi, 32, 91  
Condition variable, (*vedi* Variabili di tipo condizione)  
Condivisione dei file, 518  
Condivisione dello spazio, 483  
Condivisione falsa, 500  
Confidenzialità dei dati, 538  
Consistenza sequenziale, 501, 519, 524  
Consumo di energia, 337  
Contesto di dispositivo, 327  
Context switch, (*vedi* Cambio di contesto)  
Control Program for Microcomputer, 13  
Controllo dell'accesso obbligatorio, 604  
Controllo di comportamento, 582  
Controllo di integrità, 582  
Controllo discrezionale degli accessi, 604  
Controllore video, 320  
Controllori di dispositivi, 253  
Conversational monitor system, 54  
Convertitore analogico digitale, 427  
Copie di backup, 387  
Copy-On-Write, 223, 645, 748  
Corba, 521, 525  
Core image, (*vedi* Immagine in memoria)  
Core map, 660  
Core memory, (*vedi* Memoria a nucleo)  
Corsa critica, 91  
Co-schedulazione, 485  
CP/M, (*vedi* Control Program for Microcomputer)  
CPU, 19  
Cracker, 545  
Crittografia a chiave privata, 542  
Crittografia a chiave pubblica, 542  
CTSS, (*vedi* Compatible Time Sharing System)
- ## D
- DACL, 776  
Daemon, (*vedi* Demone)  
Data segment, (*vedi* Sezione dati)  
Deadlock, 104, 145  
Decibel, 426  
Demone, 67, 278, 636  
della stampante, 91  
di pagina, 659  
di paginazione, 223  
Denominazione uniforme, 264  
Descrittore di file, 36, 678  
Descrittore di sicurezza, 777  
Descrittori del file, 364  
Device driver, 25  
Diagrammi, 542  
Dimensione dei blocchi, 382  
Dimensione della pagina, 219  
Dining philosopher problem, (*vedi* Problema dei filosofi a cena)  
Diodo delle cattive notizie, 820  
Direct Memory Access, 28  
Directory, 35, 365  
delle pagine, 241  
di lavoro, 36, 365, 674  
di spool, 278  
operazioni, 370  
radice, 36, 365  
Directx, 718  
Diritti generici, 600  
Diritto, 591  
Dischi, 279  
CD-registrabili, 289  
CD-ROM, 285  
dischi magnetici, 279  
DVD, 291  
formattazione, 293

- memorizzazione stabile, 302  
trattamento errori, 300
- D**
- Dischi AV, 302  
Dischi dinamici, 757  
Disciplina di riga, 672  
Disk farm, 453  
Disk Operating System, 13  
Dispatcher, 80  
Disponibilità del sistema, 538  
Dispositivi a blocchi, 252, 271  
Dispositivi a caratteri, 252, 271  
Dispositivi di I/O, 252  
Dispositivo di interfaccia grafica, 327  
Dispositivo secondario, 48  
Distributed operating system, (*vedi* Sistemi operativi distribuiti)  
DMA, (*vedi* Direct Memory Access)  
Domain Name System, 513  
Dominio, 594  
Doppio interleaving, 295  
Doppio toro, 487  
DOS, (*vedi* Disk Operating System)  
Driver di dispositivi, 722  
Dropper, 570  
Dump fisico, 388  
Dump incrementale, 388  
Dump logico, 389  
DVD, 291
- E**
- Early binding, 803  
EEPROM, (*vedi* Electrically Erasable ROM)  
Electrically Erasable Rom, 23  
Elementi di controllo di accessi, 707  
Elemento della tabella delle pagine, 192  
Esperienza, ruolo nella gestione di un progetto, 820  
Estensione del file, 354  
Estensione dell'indirizzamento a finestre, 748  
Estensioni joliet, 406  
Estensioni rock ridge, 402  
Estesa, macchina, 4  
Ethernet, 479, 486, 508  
Exokernel, 55, 797
- F**
- FAT, (*vedi* Tabella di allocazione dei file)  
Fault di pagina, 188  
Fiber, 733
- FIFO, 199  
File, 353  
File condivisi, 379  
File descriptor, (*vedi* Descrittore di file)  
File di blocchi speciali, 357  
File di caratteri speciali, 357  
File mappati in memoria, 365  
File regolari, 357  
File speciali  
    a blocchi, 37  
    a caratteri, 37  
File system, 353  
    affidabilità, 386  
    backup, 387  
    caching, 394  
    CD-ROM, 399  
    chiamata di sistema UNIX, 678  
    consistenza, 391  
    CP/M, 404  
    distribuito, 518  
    FAT-12, 408  
    FAT-16, 408  
    FAT-32, 408  
    Linux, 685  
    MS-DOS, 407  
    multimedia, 440  
    prestazioni, 394  
    UNIX, 673  
    UNIX V7, 413  
    Windows 2000, 762  
    Windows 98, 410  
File system di NT, 762  
File system di rete  
    implementazione, 687  
    NFS, 687  
    protocolli, 688  
File system radice, 36  
File system strutturati a log, 398  
Finestra, 324  
Finger daemon, 585  
First Fit, 184  
Flash RAM, 23  
Font, 329  
Formattazione ad alto livello, 296  
Formattazione di basso livello, 293  
Forza bruta, 809  
Frame, 428  
Frammentazione  
    esterna, 234  
    interna, 219  
Frequenza dei fault di pagina, 217  
Funzioni di controllo VCR, 441  
Fuori linea, 7  
Furto del ciclo, 259

**G**

GDI, (*Vedi Dispositivo di interfaccia grafica*), 721  
**Gerarchia di memoria**, 173  
 Gestione dei blocchi liberi, 381  
 Gestione dei processi, UNIX, 638  
 Gestione del progetto, 817  
 Gestione dell'alimentazione,  
     Windows 2000, 721  
 Gestione della memoria, 173, 753  
     implementazione, 224  
     Linux, 662  
     liste concatenate, 183  
     mappe di bit, 182  
     memoria virtuale, 185  
     problematiche di progetto nei  
         sistemi di paginazione, 224  
     segmentazione, 229  
     swapping, 180  
     UNIX, 657  
     Windows 2000, 173  
 Gestione fault di pagina, Windows  
     2000, 751  
 Gestore degli insiemi di lavoro, 752  
 Gestore dei processi, Windows 2000, 720  
 Gestore della memoria, 173  
 Gestore della memoria, Windows 2000, 720  
 Gestore della sicurezza, Windows 2000, 720  
 Gestore di cache, Windows 2000, 721  
 Gestore di chiamate di procedure locali,  
     Windows 2000, 721  
 Gestore di configurazione, Windows  
     2000, 721  
 Gestore di file, 688  
 Gestore di oggetti, Windows 2000, 720  
 Gestore di sessione, 744  
 GID, (*Vedi Group ID*)  
 Globe, 522  
 GNU Public License, 626  
 Goat file, 579  
 GPL, (*Vedi GNU Public License*)  
 Grado di multiprogrammazione, 128  
 Grafica bitmap, 320  
 Grafica raster, 320  
 Grafica vettoriale, 320  
 Grafo diretto aciclico, 379  
 Grande spazio di indirizzamento, 821  
 Graphical User Interface, 13  
 Griglia, 487  
 Group ID, 33  
 Gruppo di cilindro, 685  
 Gruppo di processi, 638  
 GUI, (*Vedi Graphical User Interface*)

**H**

Hacker, 545  
 Handle, 70  
 Hard link, 370  
 Hard real-time system, (*vedi Sistemi  
     real-time stretti*), 135  
 Hardware di rete, 508

**I**

I/O in Windows 2000, 756  
 I/O mappato in memoria, 254  
 ID di sicurezza, 776  
 IDE, (*Vedi Integrated Drive Electronics*), 280  
 Identificatore di processo, 638  
 IEEE 1394, 30  
 IEEE standard 1003.1, 623  
 Immagine in memoria, 32  
 Implementazione bottom-up, 805  
 Implementazione top-down, 805  
 Indipendenza dal dispositivo fisico, 264  
 Indipendenza della localizzazione, 517  
 Indirezione, 808  
 Indirizzamento a blocco logico, 282  
 Indirizzo fisico, 24  
 Indirizzo IP, 512, 523  
 Indirizzo virtuale, 25  
 Industry standard architecture, 28  
 I-node, 47, 375, 681  
 Input/output, 251  
     principi hardware, 251  
 Insieme di lavoro, modello, 204  
 Instruction Set Architecture, 1  
 Integrated Drive Electronics, 26  
 Integrità dei dati, 538  
 Interfacce grafiche utente, 318  
 Interfaccia di memoria virtuale, 223  
 Interfaccia driver-kernel, 670  
 Interfaccia per le chiamate di sistema, 794  
 Interfaccia utente grafica, (*Vedi Graphical  
     User Interface*)  
 Internet, 466, 506, 521  
 Internet InterORB Protocol, 521  
 Internet Protocol, 512, 668  
 Interpretazione di comandi, (*Vedi Shell*)  
 InterProcess communication, (*Vedi  
     Comunicazione fra processi*), 91  
 Interrupt, 27  
 Interrupt vector, (*Vedi Vettore delle  
     interruzioni*)  
 Interruzione imprecisa, 263  
 Interruzione precisa, 262

Interruzioni, 260  
 Intrinseci, 332  
 Intrusi, 539  
 Invecchiamento, 133, 203  
 IP, (vedi Internet Protocol)  
 IPC, (vedi Comunicazione fra processi)  
 ISA, (vedi Instruction Set Architecture)  
 ISO 9660, 399

**J**

Java Virtual Machine, 55  
 Javaspace, 530  
 JDK 1.0, 592  
 Jiffy, 651  
 Jini, 526, 529  
     lease, 530  
     protocollo di scoperta, 529  
     servizio di ricerca, 529  
 Jitter, 424  
 Job, 6  
 JVM, (vedi Java Virtual Machine), 589

**K**

Kernel  
     modalità, 2  
     UNIX, 633

**L**

LAN, 508, 520, 529  
 Late binding, 803  
 Legacy device, 30  
 Librerie dinamiche di collegamenti, 729  
 Libro arancione, 290  
 Libro giallo, 287  
 Libro rosso, 285  
 Libro verde, 288  
 Linda, 526  
     spazio delle tuple, 526  
     tuple, 527  
 Linguaggio macchina, 1  
 Link simbolico, 380  
 Linux, 12, 625  
     file system, 685  
     gestione della memoria, 662  
     scheduling, 650  
     thread, 647

Lista concatenata, 183  
 Lista di controllo degli accessi, 596, 778  
 Livelli di software I/O, 254  
     driver dei dispositivi, 270  
     driver delle interruzioni, 268  
     indipendente dai dispositivi, 273

Località, 816

Località dei riferimenti, 204

Lock esclusivo, 677

Locking di file, 676

Login spoofing, 560

LRU, 201

Luminanza, 429

Lunghezza di dati costante, 49

Lunghezza di tempo costante, 449

**M**

Macchina a stati finiti, 81  
 Macroblocchi, 435  
 Magic number, 362  
 Mailbox, 111  
 Mainframe, 6  
 Major device, 666  
 Mappe di caratteri, 323  
 Master boot record, 296, 371  
 Master file table, 765  
 Matrice di allocazione corrente, 156  
 Matrice di richiesta, 156  
 MD5 (message digest), 543  
 Meccanismi contro politiche, 57, 799  
 Meccanismi di cache  
     multimedia, 455  
     per i file, 456  
     per i blocchi, 455  
 Meccanismi di protezione, 538  
 Meccanismo di schedulazione, 136  
 Memoria a nucleo, 22  
 Memoria ad Accesso Diretto, 28  
 Memoria associativa, 194  
 Memoria condivisa distribuita, 224, 498, 523  
 Memoria di sola lettura, (vedi Read Only Memory)  
 Memoria secondaria, 227  
 Memoria virtuale, 185  
 Message passing, (vedi Scambio di messaggi)  
 Message-passing interface, 112  
 Messaggi attivi, 495  
 Messaggio cifrato, 540  
 Messaggio in chiaro, 540  
 Microarchitettura, livello di, 1

- N**
- Microcalcolatori, 13
  - Microkernel, 56
  - Micropogramma, 1
  - Microsoft Disk Operating System, 13
  - Middleware, 508
    - basato su oggetti condivisi, 521
    - basato sul file system, 515
    - basato sulla coordinazione, 526
    - sui documenti, 514
  - MINIX, 12, 624
  - Minor device, 666
  - MMU, 186
  - Modello Bell-La Padula, 604
  - Modello di driver Microsoft, 759
  - Modo burst, 259
  - Modo cooked, 313
  - Modo fly-by, 259
  - Modo raw, 313
  - Modulazione a codice di impulsi, 427
  - Moduli caricabili, 671
  - Monitor, 104
  - Monitor degli indirizzi, 589
  - Monoprogrammazione, 174
  - Motif, 14, 332
  - Motore di mutazione, 581
  - MPI, (*vedi* Message-Passing Interface)
  - MS-DOS, (*vedi* Microsoft Disk Operating System)
    - emulazione con Windows 2000, 742
  - Multicomputer, 486
    - hardware, 486
    - interconnessione, 486
    - schedulazione, 502
    - software di comunicazione, 490
  - MULTICS, 11
  - Multiprocessore a memoria condivisa, 465, 472
  - Multiprocessore basato su directory, 472
  - Multiprocessore, sistema, 65
  - Multiprocessori, 467
    - basati su directory, 472
    - basati su interruttori Crossbar, 469
    - basati sul bus, 468
    - hardware, 468
    - NUMA, 472
    - schedulazione, 481
    - sincronizzazione, 477
    - sistemi operativi, 474
  - Multiprocessori simmetrici, 476
  - Multiprogrammazione, 9, 66
    - grado, 176
    - modello, 176
    - partizioni fisse, 175
  - Multithreading, 75
  - Mutex, 102
  - Nascondere l'hardware, 806
  - Network Operating System, (*vedi* Sistemi operativi di rete)
  - Next Fit, 184
  - NFU, 203
  - Nonpreemptive, 123
  - Notazione decimale con punti, 547
  - Notazione ungherese, 324
  - NUMA, 468
  - Numero principale di dispositivo, 274
- O**
- Object Request Brokers, 521
  - Oggetto, 521, 596
    - Windows 2000, 723
  - Oggetto di controllo, 719
  - Oggetto driver, 756
  - Open Software Foundation, 624
  - Orange book, 606
  - Orologio, 201
  - Ortegualità, 800
  - OS/360, 9
  - OSF, (*vedi* Open Software Foundation)
  - Ottimizzazione, caso comune, 817
  - Overlay, 185
- P**
- Pacchetto, 488
  - Pacchetto di riscontro, 511
  - Packet sniffer, 548
  - Pagina, 186
    - fisica, 187
    - pinning, 227
    - Paginazione, 186
      - a richiesta, 204
      - controllo del carico, 218
      - globale, 217
      - locale, 217
      - modellazione, 211
      - problematiche di progetto, 216
      - questioni implementative, 224
      - UNIX, 659
      - Windows 2000, 752
    - Pagine condivise, 221
    - Pagine di codice, 323

- Paradigma  
 dati, 792  
 esecuzione, 791  
 interfaccia utente, 791
- Paradigma event-driven, 792
- Partizione, 48
- Partizioni fisse, 175
- Password usa e getta, 551
- Path name, 36, 368
- Payload, 570
- PCI, (*vedi* Peripheral Component Interconnect)
- PDA, (*vedi* Personal Digital Assistant)
- Pendenza del cilindro, 294
- Pendenza della testina, 295
- Penetration team, 564
- Pentium, segmentazione, 235
- Perfect shuffle, 471
- Periodico, evento, 135
- Peripheral Component Interconnect, 28
- Personal Digital Assistant, 18
- PID, (*vedi* Identificatore di processo)
- Pipe, 37, 637
- Pipe symbol, 631
- Pipeline, 20, 637
- Pixel, 320, 429
- Plug and play, 30, 759
- Politica di schedulazione, 136
- Politiche contro meccanismi, 57
- Politiche di pulizia, 223
- Polling, 266
- Porta I/O, 254
- Porta nascosta, 561
- POSIX, 12, 623  
 chiamate a thread, 641
- Posizionamenti sovrapposti, 280
- Posizionamento dei file  
 multimedia, 445  
 video quasi su richiesta, 450
- Preamble, 254
- Preambolo, (*vedi* Preamble)
- Preemptive, 123
- Prepaginazione, 205
- Prestazioni, sistema operativo, 810
- Primitive asincrone, 493
- Primitive bloccanti, 493
- Primitive bloccanti e non bloccanti, 493
- Primitive non bloccanti, 493
- Primitive sincrone, 493
- Principali, 596
- Principi del software di I/O, 264  
 I/O guidato dalle interruzioni, 267  
 I/O programmato, 265  
 scopi del software, 264
- Principi di progettazione, 789
- Princípio del minimo privilegio, 601
- Princípio end-to-end, 797
- Priority inversion problem, (*vedi* Problema dell'inversione delle priorità)
- Privacy, 539
- Problema dei filosofi a cena, 113
- Problema dell'inversione delle priorità, 98
- Problema di restrizione, 609
- Process Control Block, (*vedi* Blocco di controllo dei processi)
- Process Identifier, 44
- Process table, (*vedi* Tabella dei processi)
- Processo, 32, 65  
 UNIX, 636  
 Windows 2000, 731
- Processo client, 56
- Processo leggero, 75
- Processo sequenziale, 66
- Processo server, 56
- Producer-consumer problem, (*vedi* Produttore-consommatore, problema del)
- Produttore-consommatore, problema del, 98
- Progettazione di un sistema operativo, 785
- Program Counter, 19
- Program Status Word, 19
- Prompt, 38
- Proporzionalità, 125
- Proprietà \*, 605
- Proprietà di integrità \*, 606
- Proprietà di semplice sicurezza, 605
- Protocolli di rete, 510, 512
- Protocollo, 331, 554
- Pseudoparallelismo, 65
- PSW, (*vedi* Program Status Word)
- Pull server, 441
- Punto di controllo, 158
- Punto di incrocio, 470
- Push server, 441

**Q**

- Qualità del servizio, 511
- Quanto, 130
- Quote di disco, 385

**R**

- Race condition, (*vedi* Corsa critica)
- RAID, 282  
 Livello 0, 282

- Livello 1, 283  
 Livello 2, 283  
 Livello 3, 285  
 Livello 4, 285  
 Livello 5, 285  
 striping, 283  
**RAM, (vedi Random Access Memory)**  
 RAM non volatile, 304  
 RAM video, 320  
 Random Access Memory, 20  
 Read Only Memory, 23  
 Receive, 493  
 Regione critica, 93  
 Registri di dispositivo, 2  
 Registro, 711  
 Registro base, 24, 179  
 Registro limite, 24, 179  
 Rendezvous, 112  
 Rete Omega, 470  
 Reti, 823  
 Reti di connessione a più stadi, 470  
 Reti geografiche, 508  
 Richiesta-risposta, 511  
 Rientranza, 809  
 Rifiuto del servizio, 538  
 Risorsa non prerilasciabile, 146  
 Risorsa prerilasciabile, 146  
 Risorse, 146  
 Riusabilità, 808  
 R-node, 691  
 ROM, (vedi Read Only Memory)  
 Root directory, (vedi Directory)  
 Root file system, (vedi File system radice)  
 Round, 457  
 Router, 509  
 RSA, 542  
 Rumore di quantizzazione, 427  
 RWX, bit, 38
- S**
- SACL, (vedi DACL)  
 Sandboxing, 587  
 Scambio di messaggi, 112  
 Scambio di processi, 130  
 Schedulabile, sistema real-time, 136  
 Schedulatore, 120
  - d'ingresso, 128
  - di CPU, 129
  - di memoria, 128
 Schedulazione
  - a frequenza monotona, 439
  - algoritmo di, 120
  - con priorità alla scadenza più vicina, 439
  - dei processi, multimedia, 435
  - gang, 484
  - in tempo reale, 437
  - per affinità, 482
 Schedulazione del braccio del disco, 296  
 algoritmo dell'ascensore, 298  
 first-come first-served, 297  
 shortest seek first, 297  
 Schedulazione del disco
  - dinamica, 459
  - multimedia, 457
  - statica, 457
 Scheduler, (vedi Schedulatore)  
 Scheduling
  - Linux, 650
  - UNIX, 648
  - Windows 2000, 738
 Scheduling algorithm, (vedi Schedulazione, algoritmo di )  
 Script kiddies, 549  
 Scrittore di pagine modificate, 754  
 SCSI, (vedi Small Computer System Interface)  
 Segmentazione, 230
  - implementazione, 231
  - MULTICS, 235
  - Pentium, 238
 Segmenti, 230  
 Segmento BSS, 654  
 Segmento dati, 654  
 Segmento testo, 654  
 Segnale di allarme, 32  
 Segnali di cromianza, 429  
 Semaforo, 100  
 Semaforo binario, 101  
 Semantic della condivisione dei file, 518  
 Semantica di sessione, 519  
 Send, 493  
 Separazione tra politica e meccanismo, 229  
 Sequenze di escape, 317  
 Sequenziatore, 524  
 Server di localizzazione, 526  
 Server stub, 496  
 Server X, 331  
 Servizi di rete, 511
  - datagramma, 511
  - orientati alla connessione, 511
  - orientato alla connessione, 511
  - senza connessione, 511
  - servizi con richiesta-risposta, 511
 Servizio datagramma, 511

- Servizio datagramma con messaggio di riscontro, 511  
 Settore di avvio, 743  
 SETUID, 694  
 Sezione critica, 92  
 Sezione dati, 45  
 Sezione testo, 45  
 SHA (secure hash algorithm), 543  
 Shell, 32, 38  
 Sicurezza, 537
  - chiamate API Win32, 778
  - UNIX, 692
  - Windows 2000, 775
 Sicurezza per misconoscenza, 541  
 Sicuro, 161  
 SID, (*vedi* ID di sicurezza)  
 Sincronizzazione, 102
  - multiprocessori, 477
 Singolo interleaving, 295  
 Sistema di sicurezza multilivello, 606  
 Sistema di spool, 278  
 Sistema operativo elementare per l'ingresso/uscita, (*vedi* BIOS)  
 Sistema real-time  
 Sistema real-time lasco, 18  
 Sistema real-time stretto, 18  
 Sistemi ad accoppiamento lasco, 467  
 Sistemi client-server, 798  
 Sistemi distribuiti, 506  
 Sistemi embedded, 824  
 Sistemi estensibili, 799  
 Sistemi fidati, 601  
 Sistemi multimediali, 440
  - codifica audio, 426
  - codifica video, 428
  - compressione video, 431
  - file, 425
  - paradigmi per file system, 440
  - posizionamento dei file, 445
  - schedulazione dei processi, 436
  - server, 441
 Sistemi operativi, 1
  - implementazione, 224
  - multimedia, 421
  - multiprocessore, 474
  - prestazioni, 810
  - progettazione, 785
  - struttura, 192
  - tendenze, 822
 Sistemi operativi di rete, 14  
 Sistemi operativi distribuiti, 14  
 Sistemi paralleli e distribuiti, 823  
 Skeleton, 521  
 SLED, 282  
 Small Computer System Interface, 29  
 Smart scheduling, 482  
 Socket, 667  
 Soft real-time system, (*vedi* Sistema real-time lasco)  
 Software di I/O, UNIX, 666  
 Soggetti, 596  
 Sostituzione monoalfabetica, 541  
 Sottochiave, registro, 712  
 Sottosistemi di ambiente, 728  
 Spazio dei nomi di oggetto, 726  
 Spazio di indirizzamento, 31  
 Spazio-D, 220  
 Spazio-I, 220  
 Special file, (*vedi* File speciali)  
 Spin lock, 95, 478  
 Spina-vampiro, 508  
 Spinning e switching, 480  
 Spooler directory, 91  
 Spooling, 10  
 Stack di protocolli, 512  
 Stack pointer, 19  
 Stack segment, 45  
 Stallo, (*vedi* Deadlock)  
 Standard JPEG, 433  
 Standard MPEG, 434  
 Starvation, 114, 168  
 Stateless file system, 689  
 Stato zombie, 640  
 Steganografia, 611  
 Strato di Astrazione Hardware, 716  
 Stream, 672  
 Stringa dei riferimenti, 212  
 Stringa delle distanze, 214  
 Struttura del gruppo di lavoro, 819  
 Struttura della tabella delle pagine, 192  
 Stub, 498  
 Suddivisione ampia, 455  
 Suddivisione stretta, 455  
 Suggerimenti, 816  
 Superblocco, 371  
 Superscalare, CPU, 20  
 Superuser, 33, 693  
 Supervisore, modalità, 2  
 SVID, (*vedi* System V Interface Definition)  
 Swapper, 658  
 Swapping, 180
  - UNIX, 658
 System V, 12  
 System V Interface Definition, 623

## T

Tabella dei descrittori globali, 238

Tabella dei processi, 32, 72  
 UNIX, 644  
 Windows 2000, 731

Tabella dei thread, 82

Tabella delle pagine, 188  
 a più livelli, 190  
 invertite, 196

Tabella di allocazione dei file, 374

Tabella di i-node, 681

Tasso dei fault di pagina, 215

TCB, 602

Template, 527

Tempo di binding, 802

Tempo di risposta, 125

Tendenze nella progettazione di un sistema operativo, 822

Termcap, 317

Terminale RS-232, 311

Terminale X, 331

Terminali di rete, 331  
 SLIM, 334  
 X Window, 331

Terminali orientati a carattere, 310

Text segment, (*vedi* Sezione testo)

Thread, 74  
 chiamate di sistema, 641  
 Linux, 647  
 UNIX, 646  
 Windows 2000, 737

Thread del nucleo, 799

Thread di azzeramento pagine, 755

Thread di pop-up, 87, 495

Throughput, 125

Tiger team, 564

Timer soft, 309

Timesharing, 10

Touraround, tempo di, 125

Traccia, 23

Track, (*vedi* Traccia)

Trampolino, 742

Translation Lookaside Buffer, 193  
 software per la gestione del TLB, 195

Trasparenza dei nomi, 516

Trasparenza della localizzazione, 516

Trattamento dei fault di pagina, 225

Two-phase locking, 167

**U**

UART, 312

UI, (*vedi* UNIX International)

UID, (*vedi* User ID)

UID effettivo, 694

Unità della gestione della memoria, (*vedi* MMU)

Unità integrate, elettronica per, 26

Universal Serial Bus, 29

UNIX, 619, 638  
 avvio, 652  
 Berkeley, 623  
 file system, 673  
 gestione della memoria, 654  
 I/O, 666  
 kernel, 633  
 obiettivi, 627  
 paginazione, 659  
 panoramica, 627  
 PDP-11, 621  
 portatile, 622  
 processi, 636  
 programmi di utilità, 632  
 scheduling, 648  
 shell, 629  
 sicurezza, 692  
 standard, 623  
 storia, 620  
 thread, 646

UNIX International, 624

Upcall, 86

USB, (*vedi* Universal Serial Bus)

User friendly, 14

User ID, 33

Utente, modalità, 2

Utilizzo della cache, 815

**V**

Variabili di tipo condizione, 105

Venus, 520

Verifica del codice, 562

Vettore delle interruzioni, 28, 73, 261

Vettore di risorse disponibili, 156

Vettore di risorse esistenti, 156

Vice, 520

Video interallacciato, 429

Video progressivo, 429

Video quasi su richiesta, 442  
 posizionare i file per i video quasi su richiesta, 450

Video server, 423

Video su richiesta, 423

Virtual machine monitor, 54

Virtuale, macchina, 4

Virus, 568  
 compagno, 570

- del codice sorgente, 576  
 del driver di dispositivo, 575  
 del settore d'avvio, 574  
 della macro, 575  
 delle cavità, 573  
 parassiti, 573  
 polimorfo, 581  
 residente in memoria, 573  
 sovrascriventi, 571  
**V-node**, 690
- W**
- Wakeup waiting bit, (*vedi* Bit di attesa della sveglia)  
**War dialers**, 547  
**Watchdog timer**, 308  
**Web**, 514  
**Web browser**, 514  
**Widget**, 332  
**Wildcard**, 597  
**Win32 API**, 49  
**Windows 2000**, 701  
     algoritmo di sostituzione delle pagine, 752  
     avvio, 743  
     chiamate di sistema (*vedi* Chiamate API Win32)  
     driver di dispositivo, 759  
     esecutivo, 720  
     file system, 762  
     gestione dei fault di pagina, 751  
     gestione della memoria, 173
- I/O, 756  
 kernel, 718  
 Master File Table, 765  
 oggetti, 723  
 processi, 731  
 programmazione, 709  
 registro, 712  
 scheduling, 738  
 sezioni critiche, 735  
 sicurezza, 775  
 socket, 735  
 struttura del sistema, 715  
 thread, 731
- Windows 98**, storia, 702  
**Windows me**, 14, 702  
**Windows NT**, 14, 703  
**Working directory**, (*vedi* Directory)  
**Working set**, 204  
**Worm**, 584  
**Wormhole routing**, 488  
**Worst Fit**, 184  
**WSCLOCK**, 208
- X**
- X Windows System**, 14  
**XLIB**, 332
- Z**
- Zombie**, (*vedi* Stato zombie)

